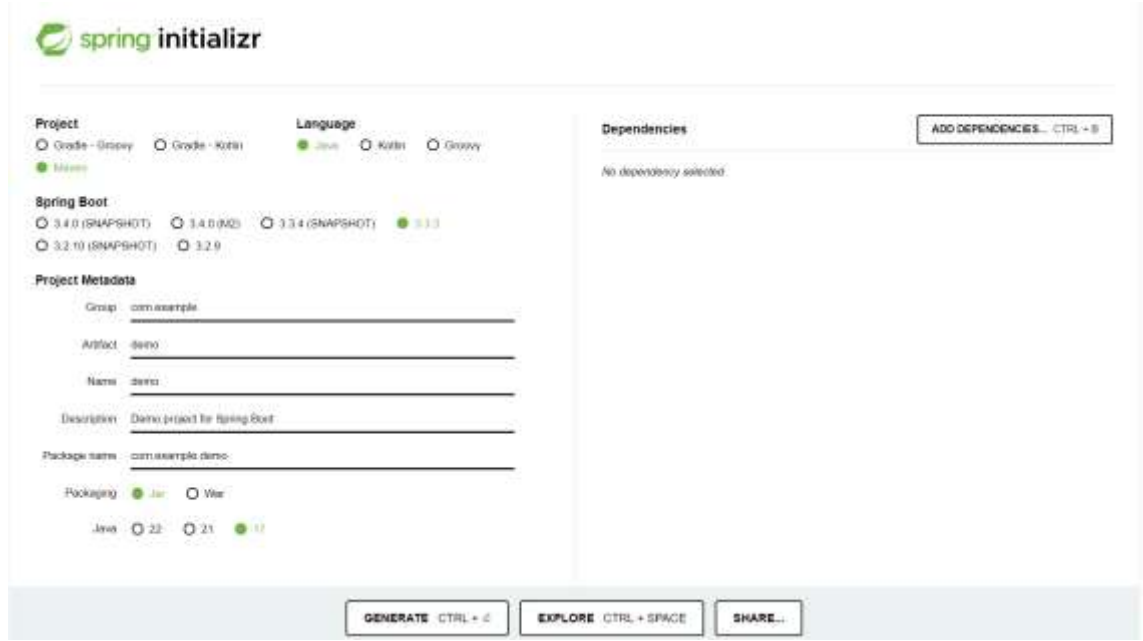


SPRING BOOT EĞİTİMİ

Spring Boot Genel Proje Yapısı

Spring Boot projesi oluşturmanın çeşitli yöntemleri bulunmaktadır. Bu dokümantasyonda, Spring Initializr ile start.spring.io sitesi yardımıyla oluşturulmuştur.



The screenshot shows the Spring Initializr web interface. It features a 'Project' section with radio buttons for 'Gradle - Groovy' and 'Gradle - Kotlin', and a 'Language' section with radio buttons for 'Java', 'Kotlin', and 'Groovy'. The 'Spring Boot' section has radio buttons for versions 3.4.0 (SNAPSHOT), 3.4.0 (M2), 3.3.4 (SNAPSHOT), 3.3.3, 3.2.10 (SNAPSHOT), and 3.2.9. The 'Project Metadata' section includes input fields for 'Group' (com.example), 'Artifact' (demo), 'Name' (demo), 'Description' (Demo project for Spring Boot), and 'Package name' (com.example.demo). There are also checkboxes for 'Packaging' (Jar, War) and 'Java' (21, 17). A 'Dependencies' section on the right has a button 'ADD DEPENDENCIES... CTRL + B' and the text 'No dependency selected'. At the bottom, there are buttons for 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

Yukarıdaki görselde bu siteden bir görünüm verilmiştir. Proje tipleri bakımından Maven ve Gradle bulunmaktadır. Java, Kotlin ve Groovy olmak üzere üç tip dil desteği vardır.

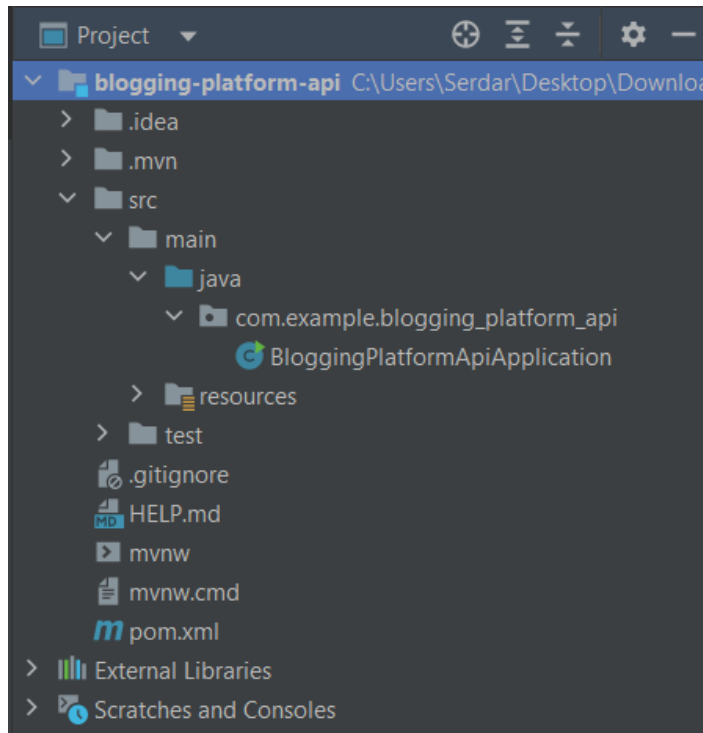
Proje metadata'sına baktığımız zaman Group kısmı, projenin bulunduğu paket adını işaret etmektedir (Genellikle burada domain isimlerinin tersi bulunur -example.com-). Artifact, projenin adıdır (Burada yapılan bir güncelleme Name kısmını da değiştirecektir.). Description bölümü, adından da anlaşılacağı üzere açıklama kısmıdır. Package name, projedeki paketlerin kaynak kodları bu kısımda tutulur (entity, controller vb.).

Dependencies kısmına bakıldığında, projeye eklenmesi istenen bağımlılıklar buradan seçilir. Genel itibariyle, projenin içeriğine ve konusuna göre eklenecek bağımlılıklar değişiklik gösterir ancak projelerde sıklıkla kullanılan belli başlı bağımlılıklar bulunur:

- 1- **Spring Boot DevTools:** Kaynak kodda yapılan deęişikliklerde otomatik olarak yeniden başlatmaya olanak sağlar.
- 2- **Lombok:** Anotasyonlarla kod üretmeyi sağlar. Boilerplate (tekrarlayan) kodları azaltmaya imkan verir. Daha temiz ve daha az kod üretimi için gereklidir.
- 3- **Spring Web:** Rest API geliştirmek, web uygulamaları oluşturmak için kullanılır.
- 4- **Spring Data JPA:** Database işlemleri için kullanılır.
- 5- **H2 Database:** Database işlemleri için kullanılan in-memory bir database'dir. Uygulama yeniden başlatıldığında database'i sıfırlar.
- 6- **MySQL/PostgreSQL Driver:** Kullanılan veritabanı için bir sürücüdür.

Bunların dışında Thymeleaf, Spring Cloud, Spring Security vb. bağımlılıklar da bulunmaktadır. Projeye göre bu bağımlılıklar deęişkenlik göstermektedir ancak yukarıda sıralı halde bulunan bağımlılıklar birçok Spring Boot projesine entegre edilirler.

Proje bağımlılıkları eklendikten sonra generate edilerek ilgili proje indirilir. İndirilen proje, seçilen IDE'de açıldıktan sonra (kullanılan IDE IntelliJ IDEA olarak belirlenmiştir) aşağıdaki gibi bir dosya yapısı görülür:



Bu dosya yapısında, **BloggingPlatformApiApplication** class'ı, Spring Boot uygulamasını başlatmak ve ayağa kaldırmak için çalıştırılacak sınıftır. **Pom.xml** dosyasında proje indirilmeden önce seçilen bağımlılıklar görülmektedir.

Proje çalıştırıldıktan sonra, konsolda aşağıdaki gibi bir çıktı alınması halinde Spring Boot projesinin ayağa kalktığı anlaşılmaktadır:

```
2024-09-13T14:55:17.164+03:00 INFO 17992 --- [Blogginq-platform-api] [ restartedMain] com.zaxxer.Hikari.HikariDataSource : HikariPool-1 - Start completed.
2024-09-13T14:55:17.164+03:00 INFO 17992 --- [Blogginq-platform-api] [ restartedMain] o.s.s.a.H2.H2ConsoleAutoConfiguration : H2 console available at '/h2-console'. Status
2024-09-13T14:55:17.164+03:00 INFO 17992 --- [Blogginq-platform-api] [ restartedMain] o.s.k.hibernate.jpa.internal.util.LogHelper : Hibernate JPA internal util LogHelper
2024-09-13T14:55:17.205+03:00 INFO 17992 --- [Blogginq-platform-api] [ restartedMain] org.hibernate.Version : Hibernate Version
2024-09-13T14:55:17.280+03:00 INFO 17992 --- [Blogginq-platform-api] [ restartedMain] o.s.s.a.internal.util.LogHelper : Second-level cache disabled
2024-09-13T14:55:17.280+03:00 INFO 17992 --- [Blogginq-platform-api] [ restartedMain] o.s.s.a.jpa.SpringPersistenceUnitInfo : No loadedDataSource setup: ignoring JPA class to
2024-09-13T14:55:17.280+03:00 INFO 17992 --- [Blogginq-platform-api] [ restartedMain] o.s.s.a.jpa.JpaPlatformInitiator : No JPA platform available (set 'hib
2024-09-13T14:55:17.280+03:00 INFO 17992 --- [Blogginq-platform-api] [ restartedMain] j.h.jpa.config.PersistenceUnitManagerFactory : Initialized JPA EntityManagerFactory for port
2024-09-13T14:55:17.280+03:00 INFO 17992 --- [Blogginq-platform-api] [ restartedMain] j.h.jpa.config.PersistenceUnitManagerFactory : spring.jpa.open-in-view is enabled by default.
2024-09-13T14:55:17.280+03:00 INFO 17992 --- [Blogginq-platform-api] [ restartedMain] o.s.s.a.jpa.internal.util.LogHelper : Listening server is running on port 3377
2024-09-13T14:55:17.280+03:00 INFO 17992 --- [Blogginq-platform-api] [ restartedMain] o.s.s.a.s.s.s.tomcat.TomcatWebServer : Tomcat started on port 8080 (http) with parame
2024-09-13T14:55:17.280+03:00 INFO 17992 --- [Blogginq-platform-api] [ restartedMain] o.s.s.BloggingPlatformApiApplication : Started BlogginqPlatformApiApplication in 4.60
```

Buradan da görüleceği üzere **Tomcat 8080** portunda ayağa kalkmıştır.

NOT: Burada, varsayılan olarak H2 database kullanımı tercih edilmiştir. MySQL, PostgreSQL gibi driverların kullanımı bir sonraki aşamada değerlendirilecektir.

Entity Class'larının Oluşturulması

Bu işlemlerin ardından, proje paketlerini oluşturmaya geçilebilir. İlk olarak, **Entity** paketi ve içerisindeki sınıflar oluşturulmalıdır. **Entity** sınıfları, çalışılan veritabanı tablolarını temsil eder. Bu dokümantasyondaki örnekte blog platformu projesi yapılacağı için **Entity** paketinin içerisine **Blog** class'ı oluşturulmuştur. Class oluşturulduktan ve veritabanındaki tabloların tanımlaması yapıldıktan sonra bu class'ın entity olabilmesi için **@Entity** anotasyonunun eklenmesi gerekir. Veritabanı tablosuna ait bir **primary key** tanımlı olduğunu belirtmek için de **@Id** anotasyonu eklenmelidir. Bu sayede bu class artık veritabanında bir tabloyu temsil edebilecektir (Tablo adı default olarak class'ın adını alacaktır, sütun isimleri de class içerisinde tanımlanmış olan field'lar olacaktır.).

```

1 package com.example.blogging_platform_api.entity;
2
3
4 import jakarta.persistence.Entity;
5 import jakarta.persistence.Id;
6
7 @Entity
8 public class Blog {
9
10
11     @Id
12     private Long id;
13     private String author;
14     private String title;
15     private String content;
16     private String category;
17 }

```

Bu bölümde tablonun oluşup oluşmadığını test etmek amacıyla projeyi çalıştırmak ve **localhost** üzerinden **h2-console**'a bağlanmak gerekir. Bunun için uygulamayı konsolda çalıştırdıktan sonra **h2-console**'da aşağıdaki ibareyi bulmak gerekir:

```

startedMain() com.fazzer.hibernate.HibernateDataSource : HibernatePool-1 - Start completed.
startedMain() a.s.b.a.H2ConsoleAutoConfiguration : H2 console available at '/H2-console'. Database available at 'jdbc:h2:mem:566b012c-d040-4615-b1fc-3b6d4fe874ce'
startedMain() o.hibernate.jpa.internal.util.LogHelper : HH000204: Processing PersistenceUnitInfo [name: default]
startedMain() org.hibernate.Version : HH000042: Hibernate ORM core version 4.5.2.Final

```

Buradaki url alındıktan sonra **localhost:8080/h2-console** adresine bağlanılıp **JDBC URL**'e konsolda görülen link yapıştırılmalıdır.

English Preferences Tools Help

Login

Saved Settings: Generic Firebird Server

Setting Name: Generic Firebird Server Save Remove

Driver Class: org.h2.Driver

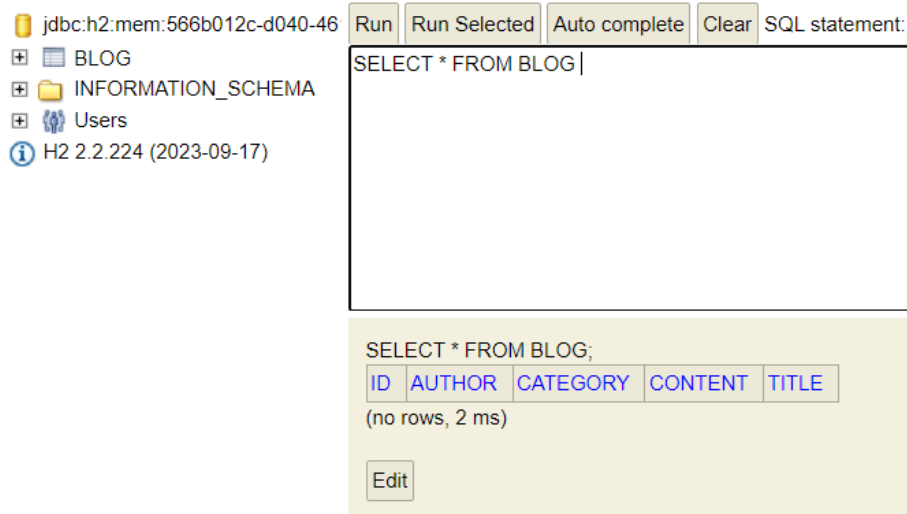
JDBC URL: jdbc:h2:mem:566b012c-d040-4615-b1fc-3b6d4fe874ce

User Name: sa

Password:

Connect Test Connection

Giriş yapıldıktan sonra IDE’de oluşturulan tablo class adını alarak sütunlarıyla beraber aşağıdaki gibi görülür.



Tablo adı default olarak class adını alır ancak bunu değiştirmek için **@Table** anotasyonu kullanılabilir. **Bunun dışında primary key değerinin artım stratejisi @GeneratedValue anotasyonu ile sağlanabilir.** Bu anotasyonun stratejilerinde birden çok tip bulunur ancak en verimli olan **SEQUENCE** seçilmiştir. Buradaki **SEQUENCE** özelliklerinin değiştirilebilmesi için **@SequenceGenerator** anotasyonu kullanılır. Mevcut sütun özelliklerini değiştirebilmek için de **@Column** anotasyonundan faydalanılır. Bu ayarlamalar yapıp program tekrar ayağa kaldırıldığında database üzerinde yapılan ayarlamalar görülecektir. Burada field’lar private tanımlanmıştır. Bunlara erişebilmek için **getter-setter** metotlarının yazılması gerekir. Bu getter-setter’lar yazılabilir ancak projeye **Lombok kütüphanesi** dahil edildiği için anotasyonlar yardımıyla da oluşturulabilir. **@Data** anotasyonu eklendiğinde tüm **getter-setter, equals, toString()** metotları oluşturulmuş olunacaktır. Toplu eklenmesi istenmezse sadece **@Getter** ve **@Setter** anotasyonları eklenerek de oluşturulabilir. Bunların hepsi eklendiğinde ilgili class’ın son görüntüsü aşağıdaki gibi olmalıdır:

```

1 package com.example.blogginy_platform_api.entity;
2
3
4 import jakarta.persistence.*;
5 import lombok.Data;
6
7 @Entity
8 @Table(name="posts")
9 @Data
10 public class Blog {
11
12     @Id
13     @SequenceGenerator(name="post_seq_gen", sequenceName = "post_gen", initialValue = 1, allocationSize = 1)
14     @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "post_seq_gen")
15     @Column(name = "ID")
16     private Long id;
17     @Column(name="TITLE", length = 100)
18     private String author;
19     @Column(name="AUTHOR", length = 50)
20     private String title;
21     @Column(name = "CONTENT", length = 300)
22     private String content;
23     @Column(name = "CATEGORY", length = 50)
24     private String category;
25
26 }

```

Ayrıca anotasyonlar yardımıyla eklenen metotlar **Structure** bölümünden de görülebilir:



Bunlarla birlikte, tek bir class'tan yönetilmesine ihtiyaç olan ve ortak olunması istenen alanlar bulunabilir (tabloya eklenen kayıtları kimin ne zaman eklediği veya kimin ne zaman güncellediği vb. bilgiler). Bunun için bir **BaseEntity** class'ı oluşturularak tek bir noktadan hepsi için ortak olan yönetimler yapılabilir. **BaseEntity** class'ına ortak olması istenen değişkenler tanımlandıktan sonra **@MappedSuperclass** anotasyonu eklenerek bu ortaklığın tüm tablolarda oluşturulması sağlanır. **Dolayısıyla BaseEntity class'ını extends eden (miras alan) tüm classlar, BaseEntity'in özelliklerinden faydalanabilecektir.** BaseEntity classı **Serializable** interface'ini implements ederek nesnenin dosyaya yazılması, veritabanına saklanması, ağ üzerinden gönderilmesi gibi işlemlerini yapmasını sağlar. Ardından **getter-setter** metotları eklenerek bu sınıf tamamlanır. Oluşturulan **Blog** class'ı, **BaseEntity'i extends** eder. Dolayısıyla **Blog class'ı da BaseEntity'i extends ettiği için Serializable özelliğini kazanmıştır ve BaseEntity'deki fieldları kendisine miras olarak almıştır.** Aşağıdaki görsellerde **BaseEntity** class'ı ve **Blog** class'ının **BaseEntity'i extends** ettiği görülmektedir.

```
public class Blog extends BaseEntity{
```

```
BaseEntity.java
1 package com.example.blogging_platform_api.entity;
2
3 import jakarta.persistence.MappedSuperclass;
4 import lombok.Getter;
5 import lombok.Setter;
6 import lombok.ToString;
7
8 import java.io.Serializable;
9 import java.util.Date;
10
11 // usage: 1 inheritor
12 @MappedSuperclass
13 @Getter
14 @Setter
15 @ToString
16 public class BaseEntity implements Serializable {
17
18     private Date createdAt;
19     private String createdBy;
20     private Date updatedAt;
21     private String updatedBy;
22 }
```

BaseEntity'de tanımlanan alanlar **Blog** class'ında **extends** edildiği için veritabanı uygulama çalıştırıldıktan sonra veritabanının son görüntüsü aşağıdaki gibi olacaktır:

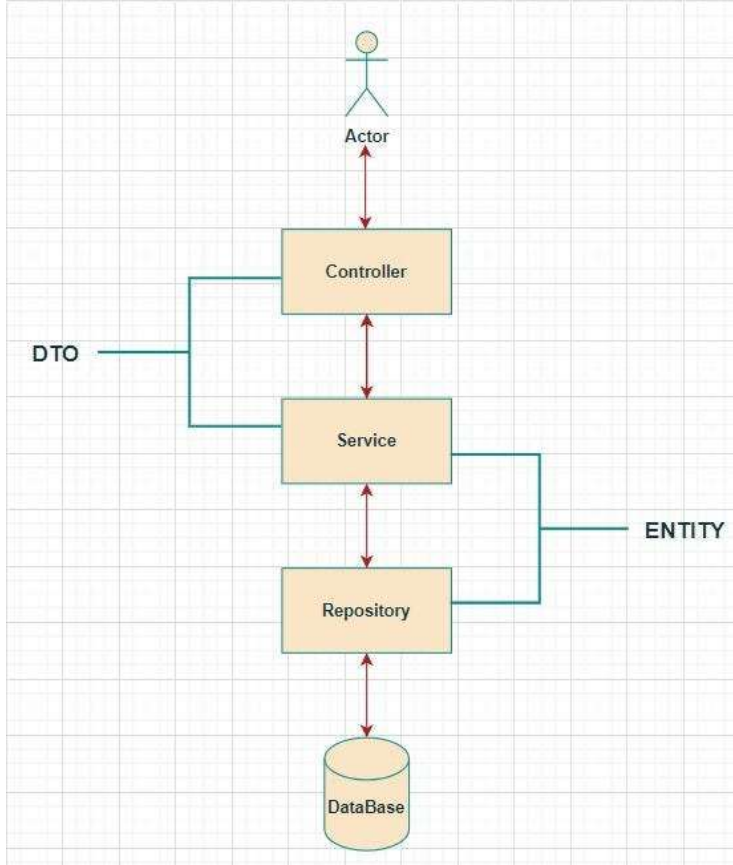


Controller, Service ve Repository Katmanları

Controller (Presentation Layer): Sunucuya gelen tüm isteklerin karşılandığı, dışarıya açılan API katmanıdır.

Service (Business Logic Layer): İş kodlarının bulunduğu, programın iş mantığının açıklandığı katmandır. Veri işleme, doğrulama ve iş mantığı bu katmanda gerçekleştirilir (*Burada bahsi geçen iş mantığı, temel iş kuralları ve operasyonlardır. Bir blog API yazılırken Service katmanında validation kontrolü yapılabilir, örneğin içerik belirli sayıda kelimeden oluşur, fazlasında hata verir. Aynı örnek üzerinden blog yazısı sayısının bir limiti olur, bu limitin aşıp aşılmadığı kontrol edilir. Bir sipariş uygulamasında stok takibi yapılır, stok varsa ürün gönderilir.*).

Repository (Data Access/Persistence Layer): Interface yardımıyla database işlemlerinin yapıldığı katmandır. JPA kullanarak database ile iletişime geçer.



Katmanlar yukarıdaki şekilde görüldüğü gibidir. Bakıldığında, **Controller** ve **Service** arasındaki haberleşme **DTO**, **Service** ve **Repository** arasındaki haberleşme **Entity** ile sağlanır.

1- Controller Katmanı

Mevcut proje üzerinden bir örnekle devam edilecek olursa, ilgili projeye **controller paketi** eklenerek **BlogController** isimli class oluşturulmuştur. Bu classın API olarak dışarıya açılabilmesi için **@RestController** anotasyonu eklenmiştir. API'nin yayınacağı adresin özelleştirilebilmesi açısından **@RequestMapping** anotasyonu eklenmiştir.

```

BlogController.java x
1  package com.example.blogging_platform_api.controller;
2
3
4  import org.springframework.web.bind.annotation.RequestMapping;
5  import org.springframework.web.bind.annotation.RestController;
6
7  @RestController
8  @RequestMapping("/blog")
9  public class BlogController {
10
11  }

```

Görselden de görüldüğü üzere **@RequestMapping** anotasyonuna eklenen ("**blog**") ibaresi, controller içerisinde yer alan **HTTP isteklerinin** ilişkilendirileceği URL'i belirtmiştir.

2- Repository Katmanı

Ardından **repository** katmanı için bir paket oluşturulmuş ve içerisine **IblogRepository** isimli interface eklenmiştir. Bu interface, **JpaRepository**'i **extends** ederek diamond operatörleri arasında iki parametre alır. Parametrelerden ilki, interface'in hangi class'a hizmet ettiğidir. İkincisi, primary key değerinin aldığı tiptir.

```

1BlogRepository.java x
1  package com.example.blogging_platform_api.repository;
2
3
4  import com.example.blogging_platform_api.entity.Blog;
5  import org.springframework.data.jpa.repository.JpaRepository;
6
7  public interface IBlogRepository extends JpaRepository<Blog, Long> {
8
9  }

```

Bu görselden de görüleceği üzere, **IblogRepository** interface'i **JpaRepository**'i **extends** etmiştir. Bu interface, **Blog** class'ı için gerekli olan tüm **CRUD işlemlerini** sağlayan bir katman görevi üstlenir.

Burada extends edilen JpaRepository, Spring Data JPA tarafından sağlanan hazır bir interface'tir ve CRUD, pagination gibi işlemleri için hazır metotlar sunar (findAll(), findById(ID id), save(Entity entity), deleteById(ID id), count() vb.).

Bu hazır metotlar dışarısında başka bir tip metot yazılmak istenirse ve istenen sorgu hazır olanlar içerisinde bulunmazsa aşağıdaki gibi bir yöntem tercih edilebilir:

```
public interface IBlogRepository extends JpaRepository<Blog, Long> {  
    Blog findByAuthor(String author);  
    Blog findByTitleAndContent(String title, String content);  
}
```

Bu görselden de görüleceği üzere arama şekli yazara veya başlık ile içeriğin birlikte olma durumuna göre ayarlanabilmektedir.

```
public interface IBlogRepository extends JpaRepository<Blog, Long> {  
    @Query("")  
    Blog getAuthor(String author);  
}
```

Benzer şekilde @Query anotasyonu kullanılarak ve içerisine istenen sorgu yazılarak özel sorgular oluşturabilmeyi sağlar.

Yukarıdaki kod görüntülerinden görüldüğü üzere Repository'de @Repository anotasyonu eklenmemiştir. Bunun nedeni, JpaRepository'nin extends edilmiş olmasıdır. Bu interface extends edildiği için Spring bunun otomatik olarak bir repository olduğunu anlar.

3- Service Katmanı

Bir sonraki adımda service katmanının oluşturulması planlanmıştır. Proje içerisinde service paketi oluşturulduktan sonra katmanlar arası bağlantılar interface'ler üzerinden yapıldığı için bir interface oluşturmak gerekir. IBlogService isimli interface, service paketinin içerisine oluşturulmuştur.

Metotlar bu interface'de tanımlanarak ve bu interface implements edilerek service paketinin içerisindeki class'a iş kodları yazılması planlanmıştır.

Bu iş kodlarının yazımı için service paketinin altında bir **impl** isimli paket daha açılarak **BlogServiceImpl** class'ı eklenmiştir. Bu class, ilgili pakette bulunan interface'i **implements** edecektir.

Bu class'ın bir **service** katmanı olduğunu belirtmek için **@Service** anotasyonu eklenmiştir.



```
1 package com.example.blogging_platform_api.service.impl;
2
3
4 import com.example.blogging_platform_api.service.IBlogService;
5 import org.springframework.stereotype.Service;
6
7 @Service
8 public class BlogServiceImpl implements IBlogService {
9
10 }
```

Katmanlar Arası Haberleşme

Bu sınıfı da oluşturduktan sonra katmanlar arası haberleşme konusuna geri dönmek gerekir. **Controller** katmanı **Service** katmanı ile ve **Service** katmanı da **Repository** katmanı ile haberleşecektir. Bundan ötürü, Controller katmanına bir property eklenmesi gerekir. Bu property'e **@Autowired** anotasyonu eklenerek **IoC (Inversion of Control)** içinde tutulan referans property'e enjekte edilmiştir. Bu anotasyon, **IBlogService** özelliğine Spring Container'dan otomatik olarak bir **IBlogService** nesnesinin enjekte edilmesini sağlar. Bu, **BlogController** sınıfının **IBlogService**'e erişmesini ve bu servisi kullanmasını sağlar.

```

1 package com.example.blogging_platform_api.controller;
2
3
4 import com.example.blogging_platform_api.service.IBlogService;
5 import com.example.blogging_platform_api.service.impl.BlogServiceImpl;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.bind.annotation.RestController;
9
10 @RestController
11 @RequestMapping("/blog")
12 public class BlogController {
13
14     @Autowired
15     private IBlogService iBlogService;
16 }

```

Bu yaklaşımın dışında **constructor injection** adı verilen bir yaklaşım da tercih edilebilir. Property'e **final** ibaresi eklenerek iki seçenek sunulur: Property'e ya ilk değer atanacak, ya da **constructor** eklenerek atama yapılacaktır. Buradaki yöntemle **constructor** eklenerek atama yapılması tercih edilmiştir.

```

1 package com.example.blogging_platform_api.controller;
2
3
4 import com.example.blogging_platform_api.service.IBlogService;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.web.bind.annotation.RequestMapping;
7 import org.springframework.web.bind.annotation.RestController;
8
9 @RestController
10 @RequestMapping("/blog")
11 public class BlogController {
12
13     1 usage
14     @Autowired
15     private final IBlogService iBlogService;
16
17     public BlogController(IBlogService iBlogService) {
18         this.iBlogService = iBlogService;
19     }
20 }

```

Buradan da görüldüğü üzere tanımlanan property'e **final** ibaresi eklenerek **constructor** üzerinden atama yapılmış ve **constructor injection** sağlanmıştır.

Benzer şekilde **Service** katmanı da **Repository** katmanı ile haberleşeceği için aynı işlemlerin burada da yapılması gerekir. **BlogServiceImpl** class'ına gidilerek **IBlogRepository** property'si oluşturulmuş (**constructor injection** yapılacaktır) ve constructor'ı bu sefer **@RequiredArgsConstructor** anotasyonu ile eklenmiştir.

```
BlogServiceImpl.java
1 package com.example.blogging_platform_api.service.impl;
2
3
4 import com.example.blogging_platform_api.repository.IBlogRepository;
5 import com.example.blogging_platform_api.service.IBlogService;
6 import lombok.RequiredArgsConstructor;
7 import org.springframework.stereotype.Service;
8
9 @Service
10 @RequiredArgsConstructor
11 public class BlogServiceImpl implements IBlogService {
12
13     private final IBlogRepository iBlogRepository;
14 }
```

Post-Put-Get-Delete Anotasyonlarının Kullanımı

RESTful web servislerindeki HTTP isteklerini işlemek için belli başlı anotasyonlar bulunur.

- 1- **@PostMapping**: HTTP yöntemi Post'tur. Yeni veri oluşturmak için kullanılır.
- 2- **@PutMapping**: HTTP yöntemi Put'tur. Veri güncellemek için kullanılır.
- 3- **@GetMapping**: HTTP yöntemi Get'tir. Veriyi almak/okumak için kullanılır.
- 4- **@DeleteMapping**: HTTP yöntemi Delete'tir. Veriyi silmek için kullanılır.

PostMapping

Bu metod **controller** içerisinde yazılır ve kullanılır. Projenin konusu itibarıyla blog yazısı ekleme işlemi yapılacağından anotasyon buna göre yazılacaktır. Metodun dışarıya açılması istendiği için **public** tanımlanmıştır. Dönüş tipi, Spring Framework'ün sağladığı **ResponseEntity** ile yapılmıştır. **Bu durum, metotlara ortak bir imza yeteneği kazandırır.** Metod, parametre olarak Blog nesnesi alacaktır. Metoda **@PostMapping** anotasyonu eklenmiştir. Bu anotasyona bir path tanımlaması yapılarak yayınacağı adres belirlenmiştir. Metodun parametre olarak aldığı nesneye **@RequestBody** anotasyonu eklenerek **JSON** nesnesi **Blog** class'larıyla eşleştirilmiştir.

```
@PostMapping("/add")
public ResponseEntity<Blog> addPost(@RequestBody Blog blog){
    return null;
}
```

Görselden de görüleceği üzere **addPost** metodu şimdilik yukarıdaki gibi tanımlanmıştır.

Bir sonraki adımda; **Controller**, **Service** ile etkileşime girdiği için **Service** katmanında da aynı metod oluşturulmuştur.

```
public interface IBlogService {

    Blog addPost(Blog blog);
}
```

Bu metod interface'e eklendikten sonra **Service** katmanı içerisinde bulunan **BlogServiceImpl** class'ı hata verecektir. Bu hatanın ortadan kalkabilmesi için ilgili metodun **@Override** edilmesi gerekir.

```
@Service
@RequiredArgsConstructor
public class BlogServiceImpl implements IBlogService {

    private final IBlogRepository iBlogRepository;

    @Override
    public Blog addPost(Blog blog) {
        return null;
    }
}
```


@Override anotasyonu; **BlogServiceImpl** sınıfındaki **addPost** metodunun, **IBlogService** interface'inde aynı isimdeki metodu geçersiz kıldığını belirtir. Bu, metodun doğru bir şekilde implements edildiğinden emin olunmasını sağlar. Eğer bu anotasyon ilgili sınıfa eklenip metod geçersiz kılınmazsa bazı uyumsuzluklar oluşabilir: Metodun imzası interface'tekinden farklı olursa kod istenilen işlevselliği sağlamaz (parametre tip uyumsuzluğu gibi). Kodun yönetimi zorlaşır, derleme zamanında hatalar ortaya çıkabilir.

Metot **@Override** edildikten sonra controller sınıfına geri dönülmelidir ve metodun içeriği doldurulmalıdır. Blog sınıfından bir **resultBlog** nesnesi oluşturulmuştur ve aynı class'ta tanımlanan **IBlogService**'indeki **addPost()** metodu çağrılarak **resultBlog** nesnesine eklenmiştir.

```
@PostMapping("/add")
public ResponseEntity<Blog> addPost(@RequestBody Blog blog){
    Blog resultBlog = iBlogService.addPost(blog);
    return null;
}
```

Ardından **BlogServiceImpl** class'ına gelerek **@Override** edilmiş metotta bazı güncellemeler yapılmıştır. Oluşturulma tarihi ve kimin oluşturulduğu set edilip ve ilgili veritabanına kaydedilerek blog nesnesi geri döndürülmüştür.

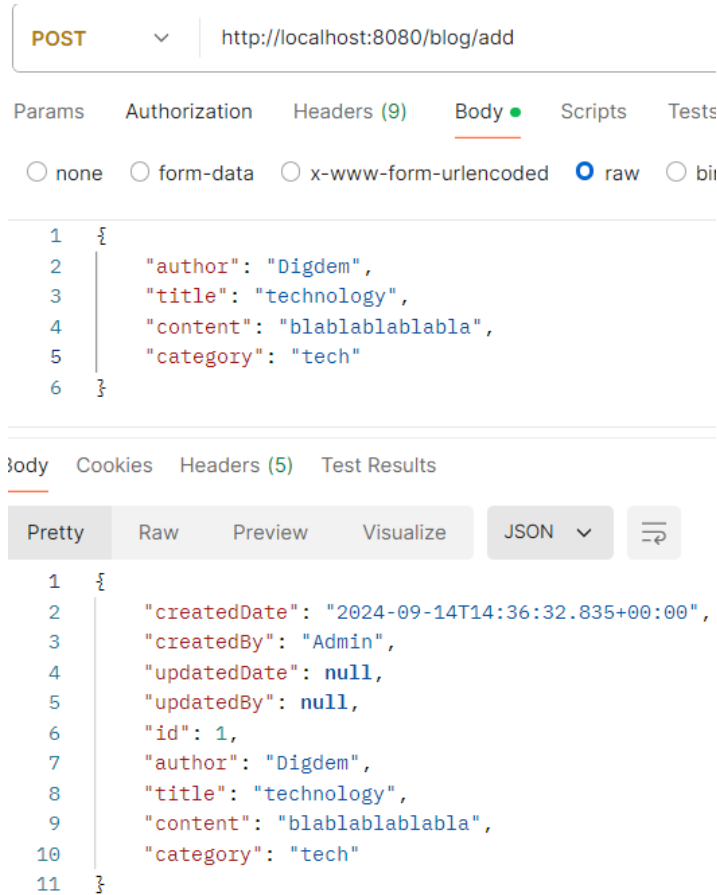
```
@Override
public Blog addPost(Blog blog) {
    blog.setCreateDate(new Date());
    blog.setCreatedBy("Admin");
    return iBlogRepository.save(blog);
}
```

Son olarak **Controller**'a tekrardan gelinerek geri dönüş tipindeki değer düzeltilmiştir. **ResponseEntity**'nin **ok()** metodunda oluşturulan kayıt döndürülmüştür.


```
@PostMapping("/add")
public ResponseEntity<Blog> addPost(@RequestBody Blog blog){
    Blog resultBlog = iBlogService.addPost(blog);
    return ResponseEntity.ok(resultBlog);
}
```

Post metodunun son hali yukarıdaki gibi olmaktadır.

Metodun doğru çalışıp çalışmadığını görmek için proje ayağa kaldırılmış ve **Postman** üzerinden **API** testi yapılmıştır.



Metot tipi **POST** olarak seçilmiş, **URL** yazılıp path'ler belirtilmiş ve raw kısmından **JSON** formatında gerekli değişkenler yazılarak doldurulmuştur. Görüldüğü üzere **API** testi başarılıdır.

GetMapping

Bu metot, **post** metodunda olduğu gibi yine **controller** içerisinde yazılır ve kullanılır. **Post** metodunda yapılan işlemler burada da tekrarlanacaktır. **Get** metodu, kayıtları almak/listelemek/okumak için kullanılır. Proje itibarıyla blog kayıtlarını okumak için bu metot kullanılacaktır. Metot dışarıya açılacağı için **public** tanımlanacak ve dönüş tipi **ResponseEntity** olarak alınacaktır. **Burada, kayıtları listelemek gibi bir durum söz konusu olduğu için generic type Blog yerine <List<Blog>> olarak alınacaktır.** Metot adı **getBlog** olarak belirlenmiştir. **@GetMapping** anotasyonu kullanılacaktır ve bu anotasyonun **path**'ine **getAll** verilmiştir. Metodun **return** ettiği değer şimdilik **null** bırakılmıştır.

```
@GetMapping("/getAll")
public ResponseEntity<List<Blog>> getBlog(){
    return null;
}
```

Bir sonraki adımda, bu metodu **service interface**'inde tanımlamak gerekir. **Metot liste şeklinde tanımlandığı için interface'te de liste tipinde almak gerekir.**

```
public interface IBlogService {

    1 usage 1 implementation
    Blog addPost(Blog blog);
    List<Blog> getAll();
}
```

Post metodunda olduğu gibi burada tanımlanan metot **service** katmanının ilgili class'ında hata verecektir. Tanımlanan metodun **@Override** edilmesi gerekmektedir. **Override edilen metotta return edilen değer iBlogRepository field'ının findAll() metodu olmalıdır.**

```
@Override
public List<Blog> getAll() {
    return iBlogRepository.findAll();
}
```

Ardından **controller** içerisinde tanımlanan metoda geri dönülerek scope'ların içi doldurulmalıdır. Bloglar liste şeklinde alınacağı için **List**'ten bir **Blog** oluşturmak ve bunu **resultBlogs** adlı değişkende tutmak gerekir. Bu değişken, **iBlogService** field'ının **getAll()** metoduyla eşlenecektir. **Return** edilecek değer **ResponseEntity**'in **ok()** metodu olmalıdır. Metodun son hali aşağıdaki gibidir:

```
@GetMapping("/getAll")
public ResponseEntity<List<Blog>> getBlog(){
    List<Blog> resultBlogs = iBlogService.getAll();
    return ResponseEntity.ok(resultBlogs);
}
```

Kodlar tamamlandıktan sonra **API** testi yapmak gerekir. **H2-database** kullanıldığı için program her çalıştığında veritabanı kendini sıfırlar. Bu yüzden mevcut kayıtların hepsini listelemeden önce veritabanına kayıt eklemek gerekir. İlgili kayıtlar **POST** metoduyla eklendikten sonra veritabanı aşağıdaki gibi görünür:

SELECT * FROM POSTS								
CREATED_DATE	ID	UPDATED_DATE	BASLIK	KATEGORI	KISI	İÇERİK	CREATED_BY	UPDATED_BY
2024-09-15 12:25:12.196	1	null	technology	tech	Digdem	biablablabla	Admin	null
2024-09-15 12:25:53.657	2	null	physics	science	Mary	i think physics is the most popular science in the world	Admin	null
2024-09-15 12:26:25.557	3	null	psychology	psych	Jane Doe	freud is the best	Admin	null

(3 rows, 4 ms)

Ardından **GET** isteği yapıp ilgili **path** girildiğinde **GET** sorgusunun doğru çalıştığı eklenen kayıtların listelenmesinden anlaşılacaktır:

```
GET http://localhost:8080/blog/getAll

Params Authorization Headers (7) Body Scripts Tests Settings
body Cookies Headers (5) Test Results
Pretty Raw Preview Visualize JSON
{
  "createdDate": "2024-09-15T09:25:12.196+00:00",
  "createdBy": "Admin",
  "updatedAt": null,
  "updatedBy": null,
  "id": 1,
  "author": "Digden",
  "title": "technology",
  "content": "blablablabla",
  "category": "tech"
},
{
  "createdDate": "2024-09-15T09:25:53.657+00:00",
  "createdBy": "Admin",
  "updatedAt": null,
  "updatedBy": null,
  "id": 2,
  "author": "Mary",
  "title": "physics",
  "content": "I think physics is the most popular science in the world",
  "category": "science"
},
{
  "createdDate": "2024-09-15T09:26:25.567+00:00",
  "createdBy": "Admin",
  "updatedAt": null,
  "updatedBy": null,
  "id": 3,
  "author": "Jane Doe",
  "title": "psychology",
  "content": "freud is the best",
  "category": "psych"
}
]
```

Kayıtlar yukarıda görüldüğü üzere başarıyla listelenmiştir.

Bakıldığında, **GET** metodu sadece tüm kayıtların listelenmesiyle gerçekleşmez. Id'ler üzerinden de sorgulama yapılabilir. Bunun için yeni bir metot yazmak gerekir. Metot **public** olarak tanımlanıp **ResponseEntity** geri dönüş tipi alacaktır. Burada tüm kayıtların listelenmesi istenmediği ve sadece id'ye yönelik arama yapılması istendiği için **List** kullanmaya gerek yoktur. Ardından metot ismi verilip parametre olarak id alınır. Metodun anotasyonuna ilgili **path** girilmelidir. Burada ufak bir ekleme yapılacaktır. **Metot, id'ye yönelik arama yapacağı için path variable olarak {id} girilmiştir. Parantez içinde tanımlanan metot parametresine @PathVariable anotasyonu eklenerek bu anotasyonuna id map'lemesi yapılmıştır. Dolayısıyla URL'den gelen id değeri burada tanımlı değişkene atılmıştır. Metot, şimdilik yine return olarak null döndürecektir.**

```

@GetMapping("/getById{id}")
public ResponseEntity<Blog> getBlogById(@PathVariable("id") Long id){
    return null;
}

```

Metodun son hali şimdilik yukarıdaki gibidir.

Diğer metotlarda olduğu gibi burada da **service interface**'inde metot tanımlanmalıdır.

```

public interface IBlogService {

    1 usage 1 implementation
    Blog addPost(Blog blog);

    1 usage 1 implementation
    List<Blog> getAll();

    Blog getBlogById(Long id);
}

```

Metot tanımlandıktan sonra service class'ında **@Override** edilmesi gerekmektedir. Bu metotta birkaç değişiklik yapmak gerekecektir. İlk olarak, **iBlogRepository** field'ının **findById(id)** metodunu çağırılmalıdır. **Bu metot bir Java class'ı olan Optional içerisinde oluşturulan bir nesnede tutulur. Nesn bulunamadığı takdirde null ile uğraşılması için kullanılır (Null Pointer Exception).** Bir if bloğu ile nesnenin var olup olmadığı kontrol edilir, varsa nesneyi döndürür ve yoksa otomatik olarak **null** döner.

```

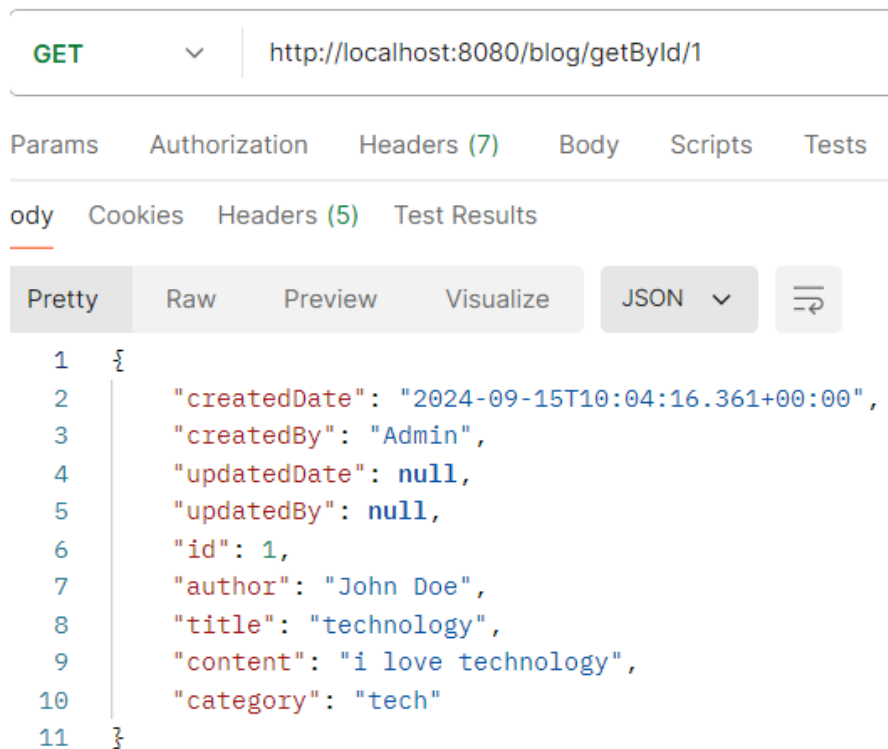
@Override
public Blog getBlogById(Long id) {
    Optional<Blog> blog = iBlogRepository.findById(id);
    if(blog.isPresent()){
        return blog.get();
    }
    return null;
}

```

Ardından **controller** katmanına geri dönerek metodun içeri düzenlenmelidir. **iBlogService** field'ının **getBlogById(id)** metodu çağırılmış ve **Blog** class'ının **getBlog** nesnesinde tutulmuştur. **Return** olarak yine **ResponseEntity**'den bu **getBlog** nesnesi döndürülmüştür.

```
@GetMapping("/getById/{id}")
public ResponseEntity<Blog> getBlogById(@PathVariable("id") Long id){
    Blog getBlog = iBlogService.getBlogById(id);
    return ResponseEntity.ok(getBlog);
}
```

Bu işlemden sonra **API** testi yapılması gerekir. Yine Postman üzerinden yapılan **API** testinin sonucu aşağıdaki gibidir (*Database'de yeniden başlatılma işlemi mevcut olduğu için öncelikle POST metoduyla kayıt eklenmeli, ardından id'ye göre çağırılmalıdır*):



Görüldüğü üzere **API** testi başarılıdır. Olmayan bir kayıt istendiğinde null döndürmektedir. İlerleyen kısımlarda bunlar için hata mesajları döndürülecektir.

PutMapping

Bu metot, diğer metotlarda olduğu gibi yine **controller** içerisinde yazılır ve kullanılır. **Get** ve **Post** metotlarında yapılan işlemler burada da tekrarlanacaktır. **Put** metodu, kayıt güncelleme işlemleri için kullanılacaktır. Metot, dışarıya açılacağı için **public** olarak tanımlanmış ve geri dönüş tipi **ResponseEntity** olarak belirlenmiştir. Metot adı **updateUser** olarak verildikten sonra parametre olarak id değeri ve bir **blog** nesnesi alınmıştır. Parametre olarak bu değerlerin alınma sebebi, id ile kaydın kontrol edilmesi ve eğer kayıt varsa var olan kaydı **blog** nesnesiyle güncellemektir. **@PutMapping** anotasyonu eklenerek ilgili **path** tanımlanmıştır. **Path** için id parametresine bir **path variable** tanımlaması yapılmıştır. **Blog** nesnesine **@RequestBody** anotasyonu eklenmiştir. **Return** değeri öncekilerde olduğu gibi şimdilik **null** olarak tanımlanmıştır.

```
@PutMapping("/edit/{id}")
public ResponseEntity<Blog> updateBlog(@PathVariable("id") Long id, @RequestBody Blog blog){
    return null;
}
```

Service interface'inde tanımlanan metot oluşturulmuştur.

```
public interface IBlogService {

    1 usage 1 implementation
    Blog addPost(Blog blog);

    1 usage 1 implementation
    List<Blog> getAll();

    1 usage 1 implementation
    Blog getBlogById(Long id);

    Blog updateBlog(Long id, Blog blog);
}
```

Bu metot, **BlogServiceImpl** class'ında **@Override** edilmelidir. Override edilen metodun içerisinde öncelikle id'ye göre kaydın var olup olmadığının kontrolü yapılmalıdır (**getBlogById()** metodundaki içerik kopyalanır.).

Kopyalanan içerikteki if bloğunda bazı değişiklikler yapılmalıdır. Id'si olan veri varsa parametre olarak gelen nesneyle güncellenmesi gerekir. **Author, title, content** ve **category** güncelleme işlemleri yapılmıştır. Güncelleyen kişi ve güncelleme zamanı set edilmiştir. Ardından blok içerisinde **iBlogRepository** nesnesinin **save** metodu çağırılarak nesne parametre olarak verilmiştir. Kayıt yoksa **null** dönecektir.

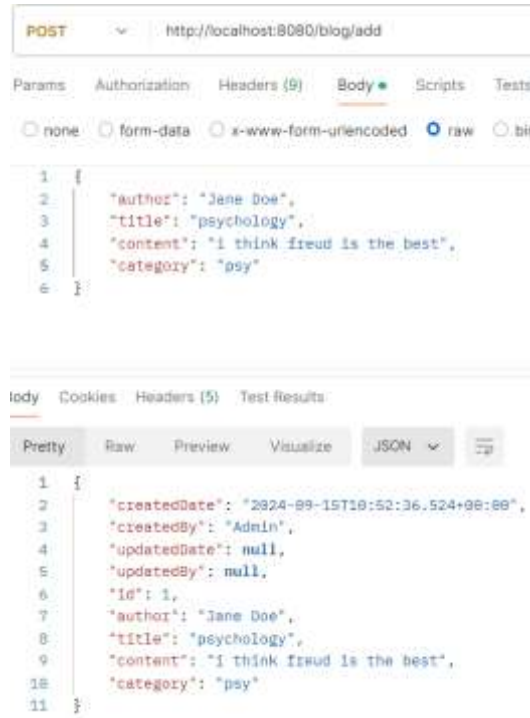
```
@Override
public Blog updateBlog(Long id, Blog blog) {
    Optional<Blog> resultBlog = iBlogRepository.findById(id);
    if(resultBlog.isPresent()){
        resultBlog.get().setAuthor(blog.getAuthor());
        resultBlog.get().setTitle(blog.getTitle());
        resultBlog.get().setContent(blog.getContent());
        resultBlog.get().setCategory(blog.getCategory());
        resultBlog.get().setUpdatedDate(new Date());
        resultBlog.get().setUpdatedBy("Admin");
        return iBlogRepository.save(resultBlog.get());
    }
    return null;
}
```

Ardından **controller** içerisine gidilerek metodun içi düzenlenmelidir. Blog class'ından **editBlog** isimli oluşturulan nesne, **iBlogService** nesnesinin **updateBlog(id,blog)** metoduna eşlenmeli ve **return** olarak **ResponseEntity**'nin **ok()** metodunda **editBlog** nesnesi döndürülmelidir.

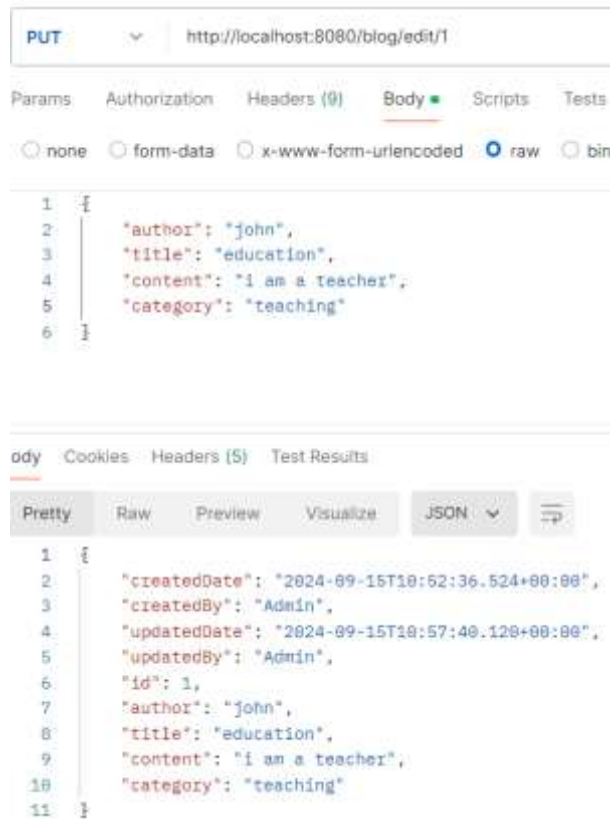
```
@PutMapping("/edit/{id}")
public ResponseEntity<Blog> updateBlog(@PathVariable("id") Long id, @RequestBody Blog blog){
    Blog editBlog = iBlogService.updateBlog(id, blog);
    return ResponseEntity.ok(editBlog);
}
```

Bu işlemten sonra metodun doğru çalışıp çalışmadığını görebilmek için **API** testi yapmak gerekir (*Database'de yeniden başlatılma işlemi mevcut olduğu için öncelikle POST metoduyla kayıt eklenmeli, güncelleme yapılmalıdır*).

Örnek olması açısından güncelleme yapılması istenen veri aşağıdaki **POST** metoduyla eklenmiştir.



İlgili içerik, aşağıdaki şekilde güncellenmiştir. **API** testi başarılıdır.



DeleteMapping

Bu metot, diğer metotlarda olduğu gibi yine **controller** içerisinde yazılır ve kullanılır. **Get**, **Post** ve **Put** metotlarında yapılan işlemler burada da tekrarlanacaktır. **Delete** metodu, kayıt silme işlemleri için kullanılacaktır. Metot, diğerlerinde olduğu gibi **public** olarak tanımlanmıştır ve dönüş tipi **ResponseEntity** alınmıştır. Bu sefer **ResponseEntity** içerisinde **Boolean** dönüş tipine sahip bir değer alınacaktır. Kayıt başarıyla silindiye **true**, silinemediyse **false** dönecektir. Metot adı belirlendikten sonra parametre olarak id alınmıştır çünkü silme işlemi id'ye göre yapılacaktır. Şimdilik dönüş tipi **null** olarak alınmıştır, sonrasında düzenlenecektir. **@DeleteMapping** anotasyonu eklenerek path tanımlaması yapılmıştır. Silme işlemi id ile yapılacağı için anotasyon id değerini de içermektedir, bu yüzden metot parametresine **@PathVariable** tanımlanmıştır. Metodun son görüntüsü aşağıdaki gibidir:

```
@DeleteMapping("/remove/{id}")
public ResponseEntity<Boolean> deleteBlog(@PathVariable("id") Long id){
    return null;
}
```

Metot, daha öncekilerde olduğu gibi **service** katmanında oluşturulmalıdır.

```
public interface IBlogService {

    1 usage 1 implementation
    Blog addPost(Blog blog);

    1 usage 1 implementation
    List<Blog> getAll();

    1 usage 1 implementation
    Blog getBlogById(Long id);

    1 usage 1 implementation
    Blog updateBlog(Long id, Blog blog);

    Boolean deleteBlog(Long id);
}
```

Daha öncekilerde olduğu gibi, **service** katmanındaki class'ta metodun **@Override** edilmesi gereklidir. Öncelikle, girilen id'ye sahip verinin var olup olmadığına bakılmalıdır. Eğer varsa **iBlogRepository** nesnesinin **deleteById(id)** metoduna erişerek veri silinmeli ve **return** değeri **true** (boolean tanımlandığı için) olmalı, eğer yoksa **return** değeri **false** olmalıdır.

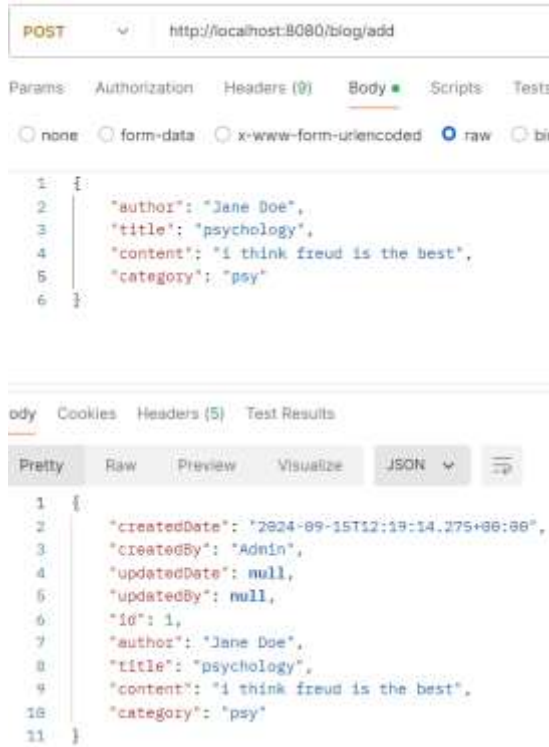
```
@Override
public Boolean deleteBlog(Long id) {
    Optional<Blog> delBlog = iBlogRepository.findById(id);
    if(delBlog.isPresent()){
        iBlogRepository.deleteById(id);
        return true;
    }
    return false;
}
```

Controller katmanındaki **deleteBlog** metoda dönülerek içerişi düzenlenmelidir. **Boolean** tipinde bir **stat** nesnesi tanımlanarak **iBlogService** nesnesinin **deleteBlog(id)** metoduna eşlenmelidir. **Return** edilerek değer **ResponseEntity**'nin **ok()** metodundaki **stat** nesnesi olmalıdır.

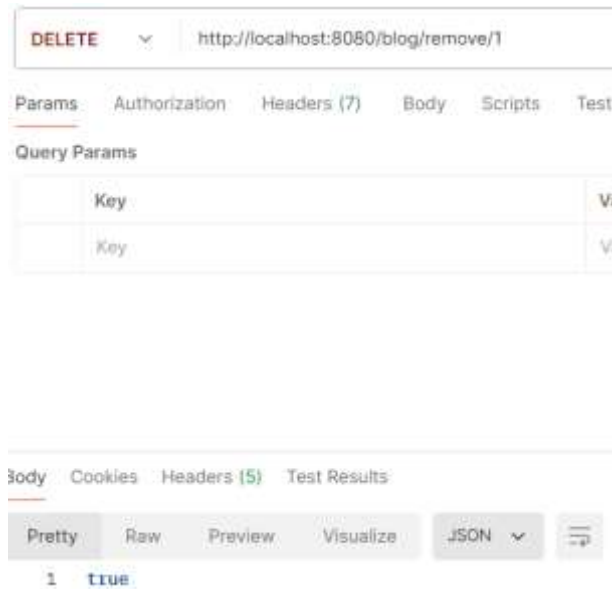
```
@DeleteMapping("/remove/{id}")
public ResponseEntity<Boolean> deleteBlog(@PathVariable("id") Long id){
    Boolean stat = iBlogService.deleteBlog(id);
    return ResponseEntity.ok(stat);
}
```

Metodun doğru çalışıp çalışmadığının kontrol edilebilmesi açısından **API** testi yapılmalıdır (*Database'de yeniden başlatılma işlemi mevcut olduğu için öncelikle POST metoduyla kayıt eklenmeli, güncelleme yapılmalıdır*).

POST metoduyla eklenen yeni veri aşağıdaki gibidir:



DELETE metoduna ilgili path girildikten ve silinmesi istenen id değeri girildikten sonra aşağıdaki çıktı alınırsa **API** testi başarılı olmuş demektir:



True değeri dönmüştür, **API** testi başarılıdır (Var olmayan bir kayıt silinmeye çalışıldığı takdirde **false** değeri dönecektir.).

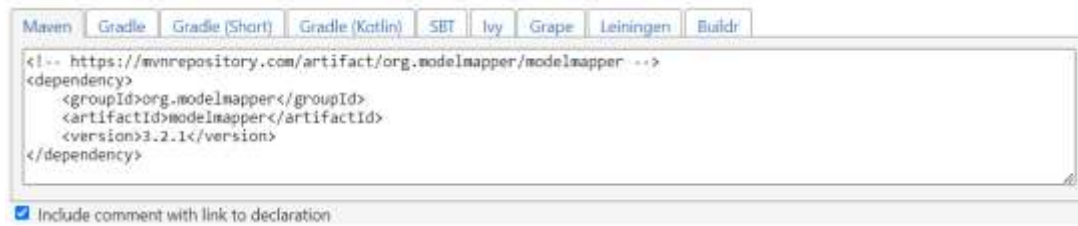
DTO (Data Transfer Object)

Veri transfer işlemi için kullanılan bir nesnedir. Veritabanından veri çekmek veya bir web hizmetine veri göndermek gerektiğinde **DTO**'lar devreye girer. Yaptığı iş veri taşımadır, herhangi bir iş mantığını barındırmaz. **DTO**'ların kullanım nedenleri:

- **Veri Transferinde Soyutlama:** Veri yapısının gereksiz detaylarını gizleyerek veri aktarımını sağlar.
- **Veri Transferini Kolaylaştırma:** İstemci ve sunucu arasında veya uygulama katmanları arasında daha az veri taşınmasını sağlar.
- **Güvenlik ve Veri Gizliliği:** **DTO**'lar, istemciden gelen veya istemciye giden verileri filtreleyebilir. Bu sayede veritabanındaki hassas bilgilerin istemciye doğrudan açılması engellenir. Örneğin, kullanıcı şifresi gibi bilgiler **DTO**'da yer almaz, böylece güvenlik artırılmış olur.
- **Validation:** Gelen verinin doğruluğunu, geçerli olup olmadığını kontrol etmek için kullanılır.
- **Performans İyileştirmesi:** Sadece gerekli verilerin taşınması performans açısından büyük bir avantaj sağlar. **DTO**'lar, gereksiz büyük veri setleri yerine sadece gereken alanları içerir. Bu sayede uygulamanın veri transfer yükü ve ağ trafiği azalır.

Entity class'ındaki bazı field'lar iç modeli ilgilendirir. Projeye bakıldığında daha öncesinde ele alınmış olan oluşturan-güncelleyen kişi ve oluşturma-güncellenme tarihi gibi field'lar iç modeli ilgilendiren alanlardır. Bunları client'a döndürmek pek iyi bir seçenek olmaz (güvenlik, veri fazlalığı vb. sebepler). Client'a sadece author, title, content ve category alanlarının döndürülmesi sağlanacaktır. Oluşturma/güncelleme konularıyla alakalı içerikler gizlenecektir.

Projenin ilerleyişi açısından bakıldığında, yapılacak ilk işlem **DTO** paketini projeye dahil etmektir. Bu paketin içerisinde **BlogDto** isimli bir class oluşturulmuştur. **Dto-Entity dönüşümünde Model Mapper kullanılacaktır.** Bu bağımlılığın projeye dahil edilebilmesi için Google'da **maven model mapper** araması yapıp en güncel olan sürüm seçilip gelecek ekrandaki bağımlılık kopyalanmalıdır.



Görselde görülen bağımlılık kopyalandıktan sonra projedeki **pom.xml** dosyasına yapıştırılmalıdır.

```
<!-- https://mvnrepository.com/artifact/org.modelmapper/modelmapper -->
<dependency>
  <groupId>org.modelmapper</groupId>
  <artifactId>modelmapper</artifactId>
  <version>3.2.1</version>
</dependency>
```

Bağımlılık projeye bu şekilde dahil edilmiş olunur. Bu adımdan sonra, uygulamada **Model Mapper**'ın instance'ını oluşturmak gerekir. Bu instance, her istenildiğinde kullanılabilir olmalıdır. Proje ayağa kalktığında bir instance'ın oluşturulup **IoC Container** içerisinde tutulması sağlanmalıdır. Kullanılmak istendiğinde **IoC Container** içerisinde referans olarak **inject** etmek gerekir. Bu konfigürasyon ayarlarının yapılabilmesi için bir **config** paketi oluşturmak ve projeye dahil etmek gerekir. Bu paketin içerisinde **ModelMapperConfig** class'ı eklenmelidir. Daha öncelerde de olduğu gibi, class'ın işlevini belirtmek için ilgili anotasyon eklenmelidir. Bu class bir konfigürasyon class'ı olduğu için **@Configuration** anotasyonu eklenmiştir. Bir sonraki adımda uygun metodu class'a oluşturmak gerekir. Geriye **model mapper** dönen **public** bir metot ve bu metot içerisinde **ModelMapper**'dan bir **instance** oluşturulmuştur. Oluşturulan **instance**'ın eşleşme stratejisi belirlenmelidir. Bu projede tam eşleştirme sağlanacaktır (**DTO**'daki field'lara karşılık gelen **Entity** nesneleri birebir örtüşen alanları eşleştirecektir.). Ardından oluşturulan **instance return** edilmelidir. Son olarak bu metoda **@Bean** anotasyonu eklenmelidir çünkü Spring'in **instance** oluşturulmasını sağlamak gerekir.

```

package com.example.blogging_platform_api.config;

import org.modelmapper.ModelMapper;
import org.modelmapper.convention.MatchingStrategies;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ModelMapperConfig {

    @Bean
    public ModelMapper getModelMapper(){
        ModelMapper modelMapper = new ModelMapper();
        modelMapper.getConfiguration().setMatchingStrategy(MatchingStrategies.STRICT);
        return modelMapper;
    }
}

```

Yukarıdaki koda bakıldığında **ModelMapper** metodu **Bean** olarak tanımlanmıştır çünkü bu metot çağrıldığında bir instance oluşturulup yönetilecek ve yeri geldiğinde **dependency injection** için kullanılacaktır.

Buradan sonra, **BlogDto** class'ına gelerek gerekli tanımlamalar yapılmalıdır. Proje gereği client'a döndürülmek istenen alanlar **author**, **title**, **content** ve **category** olarak belirlenmiştir. Oluşturma-güncelleme tarihi, oluşturan-güncelleyen kişi ve id değeri gizlenmek istenmiştir. Client'a döndürülecek field'lar **BlogDto** class'ına eklenmiştir. Ardından **@Data** anotasyonu eklenerek **getter-setter** ve **toString()** gibi metotların oluşturulması sağlanmıştır.

```

@Data
public class BlogDto {
    private String author;
    private String title;
    private String content;
    private String category;
}

```

Burada görüldüğü üzere field'lar tanımlanmış ve ilgili anotasyonla metotlar oluşturulmuştur.

Bir sonraki adımda, **DTO** dönüşümünü yapmak için **BlogController** class'ına gelinmelidir. Öncelikle **@PostMapping** anotasyonunu içeren ve yeni veri ekleme işlemini sağlayan metottan başlanmıştır. Bu metotta parametre olarak alınan **Blog** class'ı, metodun **ResponseEntity** ile vereceği geri dönüş tipi (diamond operatörleri arasındaki tanımlama) ve **resultBlog** tanımlamasının yapıldığı **Blog** class'ı **BlogDto** ile değiştirilmiştir.

```
@PostMapping("/add")
public ResponseEntity<BlogDto> addPost(@RequestBody BlogDto blog){
    BlogDto resultBlog = iBlogService.addPost(blog);
    return ResponseEntity.ok(resultBlog);
}
```

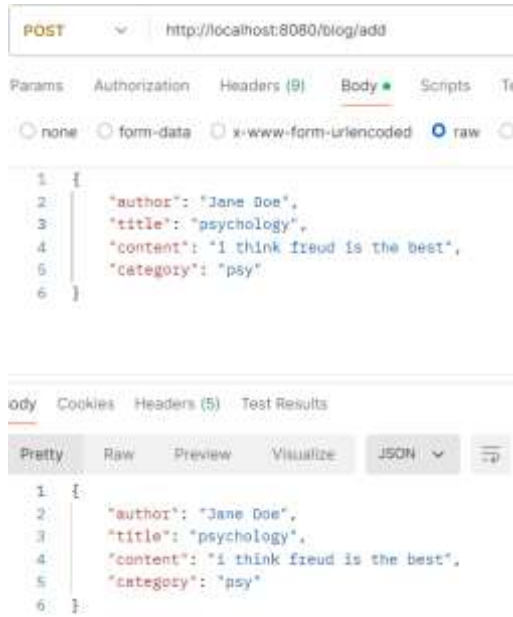
Ardından **IBlogService** interface'inde düzenleme yapılmalıdır. Metodun üretildiği ve metodun dönüş tipinin alındığı **Blog** class'ı yerine **BlogDto** alınmalıdır.

```
BlogDto addPost(BlogDto blog);
```

Ardından **BlogServiceImpl** class'ına gelerek **@Override** edilen metotta düzenlemeler yapılmalıdır ancak bundan önce **Model Mapper**'ın **IBlogRepository**'nin enjekte edildiği gibi enjekte edilmesi gerekir. Ardından metodun tanımlandığı ve geri dönüş tipini aldığı **Blog** class'ı **BlogDto** ile değiştirilmelidir. Sonra, alınan **DTO** nesnesi **Entity**'e dönüştürülmelidir. Bunun için **Blog entity**'sinden bir **instance** oluşturularak **modelMapper**'ın **map()** metoduna iki parametre verilmelidir: Dönüştürülmek istenen class ve dönüştürülecek class. Projede **BlogDto** class'ı dönüştürülmek istenen ve **Blog** class'ı da dönüştürülecek class olduğu için parametreye bu ikisi verilmelidir. **Return** edilen değer **DTO** olması gerektiği için burada da bir dönüşüm işlemi yapmak gerekir. Yeniden **modelMapper**'ın **map()** metodu kullanılarak bir öncekinde **return** edilen değer ve **BlogDto** class'ı parametre olarak verilmelidir.

```
@Override
public BlogDto addPost(BlogDto blogDto) {
    Blog blog = modelMapper.map(blogDto, Blog.class);
    blog.setCreateDate(new Date());
    blog.setCreatedBy("Admin");
    return modelMapper.map(iBlogRepository.save(blog), BlogDto.class);
}
```


Dönüşüm işleminin başarılı olup olmadığını görmek için uygulamayı çalıştırmak ve **API** testi yapmak gerekir.



Görüldüğü üzere oluşturma-güncelleme ve id ile alakalı parametreler bulunmamaktadır, sadece belirlenip eşleştirilen alanlar işlenmiştir. Aynı durumun database'deki hali aşağıdaki gibidir:

SELECT * FROM POSTS;								
CREATED_DATE	ID	UPDATED_DATE	BAŞLIK	KATEGORİ	KİŞİ	İÇERİK	CREATED_BY	UPDATED_BY
2024-09-16 13:01:48.286	1	null	psychology	psy	Jane Doe	I think freud is the best	Admin	null

Bakıldığında database'de tüm alanların tutulduğu görülür. İç modeli ilgilendiren alanlar client'a döndürülmemiştir.

Mevcut işlemler geriye kalan metotlar için de tekrar edilmelidir. **@GetMapping** anotasyonunun bulunduğu tüm verileri listeleyen metot güncellenecektir. Burada tüm verilerin listelendiği bir **Blog** yerine **BlogDto** dönecektir. Aynı şekilde metodun içerisinde bulunan **resultBlogs** nesnesinin oluşturulduğu **List** tipindeki **Blog** yerine de yine **List** tipinde **BlogDto** dönecektir.

```
@GetMapping("/getAll")
public ResponseEntity<List<BlogDto>> getBlog(){
    List<BlogDto> resultBlogs = iBlogService.getAll();
    return ResponseEntity.ok(resultBlogs);
}
```

IBlogService interface'ine gidilerek **List** tipinde **Blog** döndüren metot **BlogDto** döndürecek şekilde ayarlanmalıdır.

```
List<BlogDto> getAll();
```

BlogServiceImpl class'ına gelinerek yine **List** tipinde **Blog** döndüren metot **BlogDto** döndürmelidir. Metot **List** tipinde oluşturulduğu için **List** tipinde bir blogs nesnesi tanımlanıp daha öncesinde **return** edilen değer buraya atanmalıdır. Ardından yine **List** tipinde bir **dtos** nesnesi oluşturularak **blogs** listesini bir **stream** akışına dönüştürür. Map'leme işleminde, her bir **Blog** nesnesini **BlogDto**'ya dönüştürür. **ModelMapper** kullanarak **Blog entity**'lerini **DTO**'ya mapler. Dönüştürülen **BlogDto** nesneleri, **stream**'deki verileri bir listeye toplar. Metodun sonunda **dtos** nesnesi **return** edilir.

```
@Override
public List<BlogDto> getAll() {
    List<Blog> blogs = iBlogRepository.findAll();
    List<BlogDto> dtos = blogs.stream().map(blog -> modelMapper.map(blog, BlogDto.class)).collect(Collectors.toList());
    return dtos;
}
```

Bu kod bloğundan da görüleceği üzere bir lambda ifadesi kullanılmıştır. **Bu lambda ifadesi, her bir blog nesnesini alır ve ModelMapper ile onu BlogDto'ya dönüştürür.**

DTO dönüşümünün başarılı olup olmadığını görmek adına **API** testi yapmak gerekir.



Yukarıdaki görsele göre **API** testi başarılıdır.

Bir sonraki adımda, yine **@GetMapping** isteğini bu sefer id'ye göre yapan metodun düzenlenmesi ve **DTO** dönüşümünün yapılması gerekir. Bunun için **controller** class'ında bulunan metotta dönüş tipini ve nesnenin tanımlandığı class'ı **Blog**'dan **BlogDto**'ya dönüştürmek gerekir.

```
@GetMapping("/getById/{id}")
public ResponseEntity<BlogDto> getBlogById(@PathVariable("id") Long id){
    BlogDto getBlog = iBlogService.getBlogById(id);
    return ResponseEntity.ok(getBlog);
}
```

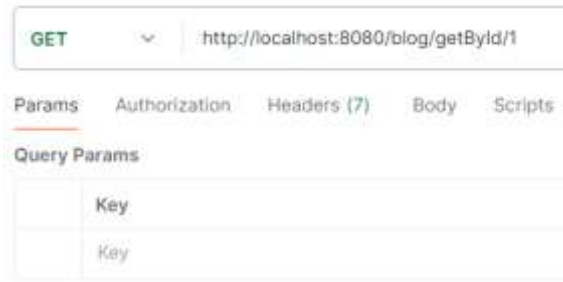
IBlogService interface'inde metot **Blog** yerine **BlogDto** class'ından tanımlanmalıdır.

```
BlogDto getBlogById(Long id);
```

BlogServiceImpl class'ında **@Override** edilmiş metotta **BlogDto** tanımlaması yapılmalıdır. If bloğu içerisindeki **return** edilen değere map'leme işlemi yapılmalıdır.

```
@Override
public BlogDto getBlogById(Long id) {
    Optional<Blog> blog = iBlogRepository.findById(id);
    if(blog.isPresent()){
        return modelMapper.map(blog.get(), BlogDto.class);
    }
    return null;
}
```

Metodun **DTO** dönüşümünü doğru şekilde yapıp yapmadığından emin olmak için yine **API** testi yapılması gerekir.



Bakıldığında **API** testinin başarılı olduğu görülmektedir.

Bir sonraki metotta **@PutMapping** işlemini yapan ve güncelleme işlevi gören metotta **DTO** dönüşümü yapılmalıdır. Metodun dönüş tipi ve **editBlog** nesnesinin üretildiği class olan **Blog**, **BlogDto** ile değiştirilmelidir.

```
@PutMapping("/edit/{id}")
public ResponseEntity<BlogDto> updateBlog(@PathVariable("id") Long id, @RequestBody BlogDto blog){
    BlogDto editBlog = iBlogService.updateBlog(id, blog);
    return ResponseEntity.ok(editBlog);
}
```

IBlogService interface'ine gidilerek **Blog** ile **BlogDto** değiştirilmelidir.

```
BlogDto updateBlog(Long id, BlogDto blog);
```

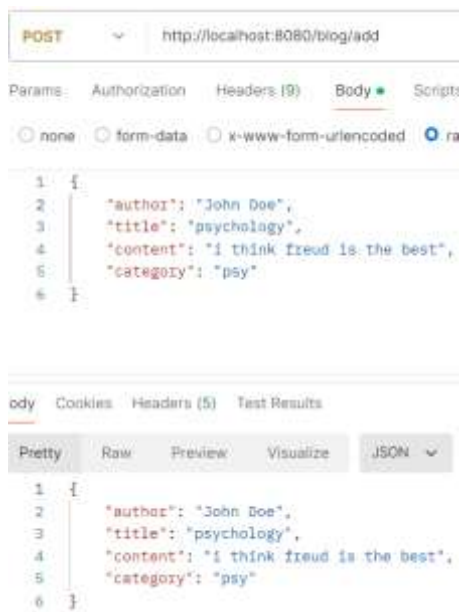
BlogServiceImpl class'ına **@Override** edilen metotta düzenlemeler yapılmalıdır. **Blog** ile tanımlı metot ve aldığı parametre, **BlogDto** ile değiştirilmelidir. **Return** edilen değer üzerinde **map**'leme işlemi yapılmalıdır.

```

@Override
public BlogDto updateBlog(Long id, BlogDto blog) {
    Optional<Blog> resultBlog = IBlogRepository.findById(id);
    if(resultBlog.isPresent()){
        resultBlog.get().setAuthor(blog.getAuthor());
        resultBlog.get().setTitle(blog.getTitle());
        resultBlog.get().setContent(blog.getContent());
        resultBlog.get().setCategory(blog.getCategory());
        resultBlog.get().setUpdatedDate(new Date());
        resultBlog.get().setUpdatedBy("Admin");
        return modelMapper.map(IBlogRepository.save(resultBlog.get()), BlogDto.class);
    }
    return null;
}

```

Metottaki **DTO** dönüşümü **API** ile test edilmelidir.



Eklenecek ve güncellenmek istenen kayıt yukarıdaki gibidir.

PUT

▼

http://localhost:8080/blog/edit/1

Params

Authorization

Headers (9)

Body ●

☐ none

☐ form-data

☐ x-www-form-urlencoded

```
1 {
2   "author": "jane",
3   "title": "education",
4   "content": "i am a teacher",
5   "category": "teaching"
6 }
```

Body

Cookies

Headers (5)

Test Results

Pretty

Raw

Preview

Visualize

JSON

```
1 {
2   "author": "jane",
3   "title": "education",
4   "content": "i am a teacher",
5   "category": "teaching"
6 }
```

Kaydın güncelleme işlemi başarılıdır.

Görüldüğü üzere **@DeleteMapping** anotasyonunun sağladığı kayıt silme işlemi için herhangi bir DTO dönüşümü yapılmamıştır. Bunun nedeni, silme işleminin veri döndürmemesinden kaynaklıdır. Sadece boolean bir ifade döner. Silinen kaydın DTO dönüşümünü yapmak bu yüzden anlamsızdır.

Bu projede DTO dönüşümü için **ModelMapper** kullanılmıştır ancak DTO dönüşümü sadece **ModelMapper** ile yapılmaz. Manuel dönüşümler, **MapStruct**, **Dozer**, **Selma**, **Orika** vb. yöntemler de bulunmaktadır.
