

Proje Detay Raporu

Java Task Sunum

Javada bir yayınevi yönetiminin bir prototipi

Hazırlayan
Ömer Faruk Kan

Bu süreçte dikkat edilmesi gerekenler:

- Clean Code
- Documentation
- Modülerlik
- SOLID prensipleri
- MVC yapısı

Bu proje raporu, yazılım geliştirme sürecinde temel prensipleri ön planda tutarak ilerlediğimiz bir adımdır. MVC mimarisini kullanarak, temiz kod yazımı, ayrıntılı dokümantasyon, modülerlik ve SOLID prensiplerine uygunluk gibi kritik unsurlara odaklanarak bir yazılım çözümü geliştirdik. Ayrıca, bağımlılıkları en aza indirgeyerek ve generic yapılarla sürdürülebilir bir kod tabanı oluşturarak projenin kalitesini ve yönetilebilirliğini artırdık. Aynı zamanda gerekli yerlerde yorum satırları ile kod anlaşılabilirliğine katkıda bulunduk. Bu rapor, bu süreçte benimsediğimiz yaklaşımları ve elde ettiğimiz sonuçları ayrıntılı bir şekilde sunacaktır.

Müşteri gereksinimlerini karşılamak için geliştirdiğimiz yazılım çözümü, bir yayınevinin işlevselliğini sağlamak üzere tasarlanmıştır. Bu çerçevede, yayınevlerini, kitapları ve yazarları yönetmek için temel CRUD (Create, Read, Update, Delete) işlemlerini destekler. Ayrıca, belirli işlevleri gerçekleştirmek için özelleştirilmiş taleplere yanıt verebilecek şekilde tasarlanmıştır.

Yazılımın özelliklerini kullanıcıların anlaması ve test etmeleri için Swagger ve Postman gibi popüler API test araçlarıyla uyumludur. Bu sayede, kullanıcılar kolayca API çağrılarını yaparak işlevleri kontrol edebilir ve doğrulayabilirler.

Verilerin saklanması için metin dosyaları kullanılmıştır. Her bir dosya, ilgili varlık türüne (yayınevi, kitap, yazar) ait bilgileri içerir. Bu yapı, veri yönetimini basitleştirir ve verilerin kolayca erişilebilir olmasını sağlar.

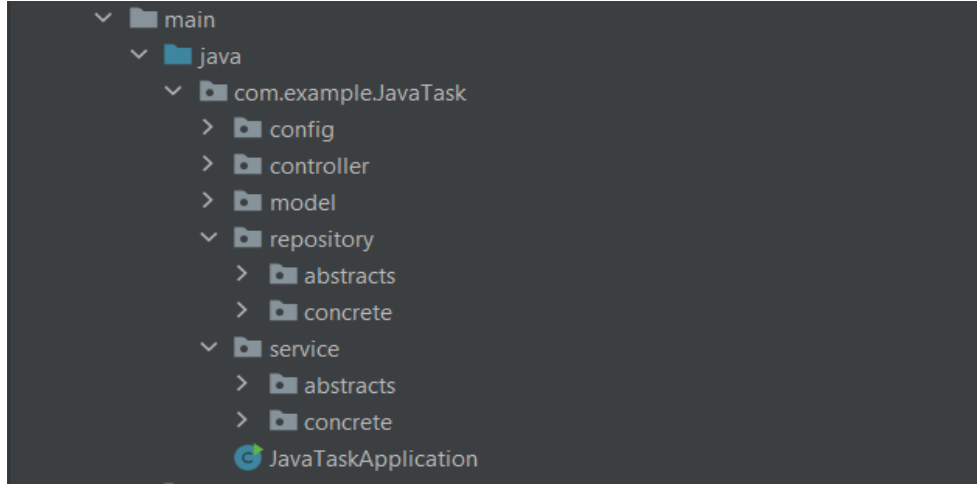
İlerleyen aşamalarda, metin dosyaları yerine veritabanı gibi daha güçlü ve ölçeklenebilir veri depolama çözümleri kullanılabilir. Bu, uygulamanın performansını artırabilir ve daha karmaşık veri manipülasyonlarına olanak tanır.

Sonuç olarak, geliştirilen yazılım çözümü müşterinin ihtiyaçlarını karşılayacak şekilde tasarlanmış ve uygulanmıştır. CRUD işlevselliği, veri yönetimi ve API uyumluluğu gibi temel özellikler, müşterinin beklentilerini karşılamak için başarıyla entegre edilmiştir.

Ayrıca, geliştirilen yazılım çözümü için kapsamlı bir JUnit test paketi oluşturulmuş ve başarıyla uygulanmıştır. Bu testler, yazılımın farklı bileşenlerinin doğruluğunu ve güvenilirliğini doğrulamak için tasarlanmıştır. Tüm testler sorunsuz bir şekilde geçilmiş ve beklenen davranışları doğrulamıştır. Bu başarı, yazılımın sağlamlığı konusunda bize fikir verir.

[Projemizin UML sınıf diyagramı](#), sistemin bileşenlerini ve aralarındaki ilişkileri görselleştirerek proje mimarisini açıkça ortaya koyuyor. Bu diyagram, projenin katmanlarını, sınıflarını ve bu

sınıflar arasındaki ilişkileri detaylı bir şekilde tanımlıyor. Sınıf diyagramındaki bileşenlerin her biri, projenin farklı modüllerini ve işlevlerini temsil ediyor. Bu sayede, proje ekibimiz ve paydaşlarımız, projenin genel yapısını daha iyi anlayabilir ve işbirliği daha etkili hale gelebilir. Genel olarak proje mimarisine bakabiliriz.



Projenin dizin yapısı görülmektedir.

Model sınıfları

MVC (Model-View-Controller) mimarisi, birçok modern web uygulamasında kullanılan popüler bir tasarım desendir. Bu mimaride, veri işleme (Model), kullanıcı arayüzü (View) ve uygulama mantığı (Controller) bileşenleri birbirinden ayrılmıştır. Spring Framework'u kullanan birçok uygulama, bu MVC mimarisini benimser ve bu yapıyı kullanarak uygulama geliştirir.

Model bileşeni, uygulamanın veri işleme işlevselliğini temsil eder. Veritabanı işlemleri, veri işleme, doğrulama ve veri manipülasyonu gibi görevler bu bileşen altında gerçekleştirilir. Spring projelerinde, bu tür veri erişimi işlemleri genellikle repository sınıfları aracılığıyla gerçekleştirilir.

Model sınıfları, bir projenin veri yapısını, iş mantığını ve veri işleme işlevlerini temsil eder. Bu sınıflar, projenin düzenli ve düzenlenmiş bir şekilde çalışmasını sağlar ve veri manipülasyonunu kolaylaştırır.



Book sınıfı, bir kitabı temsil eder ve kitapların özelliklerini içerir. Bu özellikler arasında kitap kimliği (bookId), başlık (title), fiyat (price), ISBN numarası (isbn) ve yayınevi kimliği

(publisherId) bulunur. Jakarta'nın (Java EE'nin) JPA standartlarını kullanarak @Entity anotasyonu ile işaretlenmiş ve veritabanında bir tablo olarak temsil edilir. Ayrıca, Lombok kütüphanesini kullanarak otomatik olarak getter, setter, equals ve hashCode metodlarını oluşturur, bu da kodun daha temiz ve okunabilir olmasını sağlar. Book sınıfı, projemizin diğer bileşenleriyle etkileşim kurarak kitapların veritabanında depolanmasını ve işlenmesini sağlar.

```
@Entity
@AllArgsConstructor
@NoArgsConstructor
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int bookId;
    private String title;
    private double price;
    private String isbn;
    private int publisherId;
```

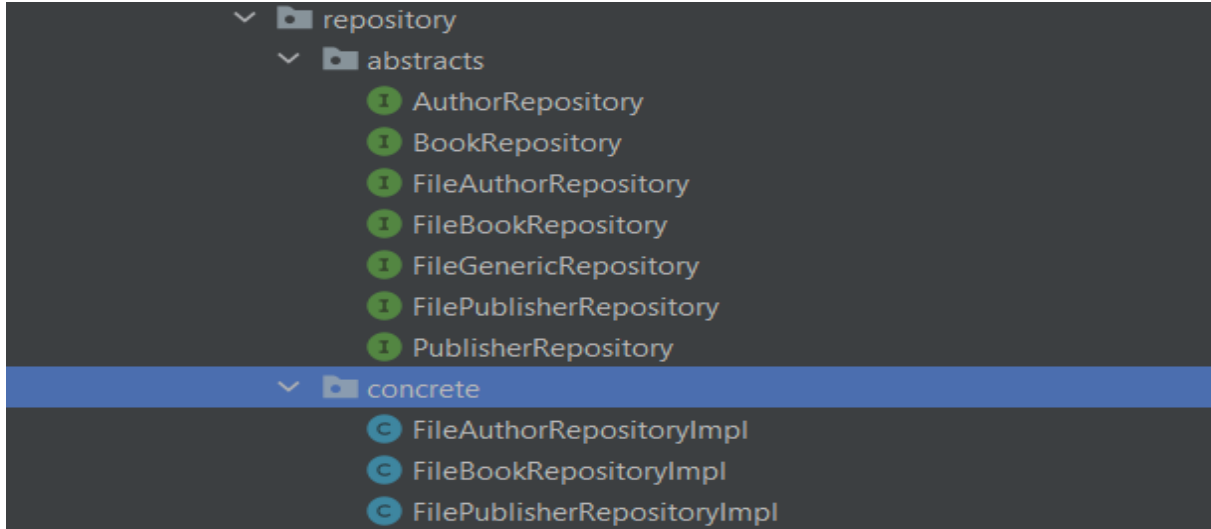
Publisher sınıfı, bir yayınevinin temsil eder ve yayınevlerinin özelliklerini içerir. Her bir yayınevinin bir yayınevi kimliği (publisherId) ve bir yayınevi adı (publisherName) bulunur. @Id anotasyonu, Publisher sınıfında kullanılarak publisherId alanının birincil anahtar alanı olduğunu belirtir. Bu, her yayınevinin benzersiz bir kimliğe sahip olması gerektiğini ve bu kimliğin bu alan aracılığıyla sağlandığını gösterir. @GeneratedValue(strategy = GenerationType.IDENTITY) anotasyonu ise publisherId alanının otomatik olarak artan bir değer olarak oluşturulacağını belirtir. Bu sayede, her yeni yayınevi kaydedildiğinde, veritabanı otomatik olarak bir sonraki benzersiz kimliği atar. Bu strateji genellikle ilişkisel veritabanlarında kullanılır ve birincil anahtar alanının otomatik olarak artan bir değer olmasını sağlar. Publisher sınıfı, projenin diğer bileşenleriyle etkileşim kurarak yayınevlerinin veritabanında depolanmasını ve işlenmesini sağlar.

```
@Entity
@AllArgsConstructor
@NoArgsConstructor
public class Publisher {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int publisherId;
    private String publisherName;
```

Author sınıfı, kitap yazarlarını temsil eder ve yazarların özelliklerini içerir. Her bir yazarın bir yazar kimliği (authorId), adı (firstName), soyadı (lastName) ve ilgili kitabın kimliğini temsil eden bir kitap kimliği (bookId) bulunur. Bu sınıf da, veritabanında bir tablo olarak temsil

edilmesi için @Entity anotasyonu ile işaretlenmiştir. Yine aynı şekilde Lombok kütüphanesi kullanılmıştır @Id ve @GeneratedValue annotation kullanılmıştır Author sınıfı, projenin diğer bileşenleriyle etkileşim kurarak yazarların veritabanında depolanmasını ve işlenmesini sağlar

Repository Sınıfları



Repository sınıfları, veritabanı işlemlerini yürütmek ve veriye erişmek için kullanılır. Bu sınıflar, doğrudan veritabanı ile iletişim kurar ve uygulama katmanları arasında veri akışını sağlar. Spring Data JPA gibi teknolojilerle birleştirildiğinde, repository sınıfları veri erişimini büyük ölçüde basitleştirir ve veritabanı işlemlerinin yazılmasını kolaylaştırır.

Repository sınıfları, genellikle CRUD (Create, Read, Update, Delete) işlemlerini gerçekleştiren metodları içerir. Bu metodlar, uygulamanın farklı bileşenlerinden veriye erişim sağlamak için kullanılır. Ayrıca, bu sınıflar genellikle JPA (Java Persistence API) entegrasyonu ile birlikte kullanılır ve veritabanı işlemlerini nesne ilişkilendirme teknikleriyle gerçekleştirir.

Bu nedenlerden dolayı, repository sınıfları Spring projelerinde önemli bir rol oynar. Veritabanı işlemlerini soyutlar, kod tekrarını azaltır ve veri erişimini yönetmek için bir arayüz sağlar. Bu sayede, uygulamanın veri işleme katmanı daha modüler, esnek ve bakımı kolay hale gelir.

Projemizde şuanlık metin dosyalarında tutulan verilerle çalışabilmek ve CRUD işlemlerini yapabilmek için dosya işlemleri ile algoritmik CRUD işlemlerini gerçekleştirir. FileRepository olarak adlandırdığımız sınıflarda herhangi bir java kütüphanesi kullanılmadan elle yazılmıştır. Bu şekilde bir yapı tasarlanmasının sebebi kod anlaşılabilirliğini ve modülerliği sağlamak. Bu sayede servis sınıflarında kod karmaşası önlenmiş olup daha sürdürülebilir bir yapı sağlanmaktadır. Aynı zamanda java generic yapılar sayesinde Bağımlılık en aza indirgenmiştir. Bu sayede hiç bir servis sınıfı asıl işi yapan repository sınıflarından habersiz bir şekilde interfacerler aracılığı ile haberleşmektedir. Generic yapıların diğer bir avantajı ise veri tipi uyumsuzluklarının önlenmesi ve performans artışı olmuştur. Aynı zamanda gereksiz

kod tekrarı bu şekilde önlenmiştir. Örnek vermek gerekirse FilePublisherRepository sınıfının sadece kendine ait metoda sahip olması diğer repository sınıflarını etkilemez.

```
public interface FileGenericRepository<T> {  
    3 implementations  
    List<T> getAll();  
    3 implementations  
    void save(T object);  
    3 implementations  
    T getById(int id);  
    3 implementations  
    void update(T object);  
    3 implementations  
    void delete(int id);  
}
```

```
6 pages 1 implementation  
public interface FileAuthorRepository extends FileGenericRepository<Author>{  
  
}
```

```
1 page 1 implementation  
public interface FileBookRepository extends FileGenericRepository<Book> { }
```

```
1 page 1 implementation  
public interface FilePublisherRepository extends FileGenericRepository<Publisher>{  
    1 usage 1 implementation  
    List<String> listBooksAndAuthorsOfTwoPublishers();  
}
```

Proje dizin yapısında soyut sınıflar ile concrete sınıflar birbirinden ayrılarak modülerlik desteklenmiştir. Concrete repository sınıfları tamamen kendilerine ait soyut sınıflardan implemente edilerek dosya işlemleri ile CRUD işlemlerini gerçekleştirir.

```

@Repository
public class FileAuthorRepositoryImpl implements FileAuthorRepository {

    4 usages
    private final String filePath;

    1 usage
    public FileAuthorRepositoryImpl() {
        this.filePath= "C:\\Dev\\JavaTask\\src\\DB\\Yazar.txt";
    }

    @Override
    public List<Author> getAll() {
        List<Author> authors = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(regex: ",");
                Author author = new Author();
                author.setAuthorId(Integer.parseInt(parts[0]));
                author.setFirstName(parts[1]);
                author.setLastName(parts[2]);
                author.setBookId(Integer.parseInt(parts[3]));
                authors.add(author);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return authors;
    }
}

```

Bu metod, tüm yazarları bir liste olarak döndürmek için kullanılır. İşlevselliği gereği, projenin dosya tabanlı bir veri saklama yöntemi kullandığı varsayılarak, belirli bir dosyadan yazar verilerini okur. Metod, parametre olarak bir dosya yolunu alır ve bu dosyadaki her satırı bir yazar nesnesine dönüştürür. Her satır, belirli bir desenle ayrılmış bir formatta bulunan yazar özelliklerini içerir. Satır okunduğunda, ',' karakterine göre bölünür ve parçalar ilgili yazar özelliklerine atanır. Daha sonra, bu özelliklere sahip olan yazar nesnesi, oluşturulan yazar listesine eklenir. Metod, IOException olası bir dosya okuma hatası durumunda bir hata izi (stack trace) oluşturarak hatayı işler. Son olarak, bu metod, yazarların bulunduğu bir liste döndürür.

```

@Override
public void save(Author author) {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath, append: true))) {
        writer.write( str: author.getAuthorId() + ";" + author.getFirstName() +
            ";" + author.getLastName() + ";" + author.getBookId());
        writer.newLine();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

Bu metod, yeni bir yazarın verilerini dosyaya kaydetmek için kullanılır. Metod, bir Author nesnesi alır ve bu nesnenin özelliklerini bir dosyaya yazarak yazarın verilerini depolar. Dosya işlemleri için Java'nın standart giriş-çıkış (I/O) kütüphanesi kullanılmıştır.

Metod, parametre olarak bir Author nesnesi alır ve bu nesnenin özelliklerini kullanarak bir satır oluşturur. Oluşturulan satır, yazarın kimlik numarası (authorId), adı (firstName), soyadı (lastName) ve kitap kimliği (bookId) bilgilerini içerir. Bu bilgiler ';' karakteriyle ayrılmış bir formatta yazılır.

Dosyaya yazma işlemi, try-catch bloğu içinde gerçekleştirilir. Bu sayede, dosya işlemleri tamamlandıktan sonra otomatik olarak kaynaklar serbest bırakılır ve bellek sızıntısı önlenir. Dosya yazma işlemi sırasında oluşabilecek IOException türündeki hatalar, try-catch bloğu içinde işlenir ve hatanın stack trace'i konsola yazdırılır.

Bu metod, yeni bir yazarın verilerini dosyaya kaydetmek için kullanılan temel bir işlevselliği sağlar. Dosya işlemleri, projenin dosya tabanlı bir veri saklama yöntemi kullandığı varsayılarak gerçekleştirilir.

```

@Override
public Author getById(int authorId) {
    List<Author> authors = getAll();
    for (Author author : authors) {
        if (author.getAuthorId() == authorId) {
            return author;
        }
    }
    return null; // Yazar bulunamadı
}

```

Bu metod, belirli bir yazar kimliği (authorId) ile eşleşen yazarı döndürmek için kullanılır. İşlevselliği gereği, belirli bir yazar kimliğiyle ilgili tüm yazarları içeren bir liste alır ve bu listede döngü yaparak belirli bir kimliğe sahip yazarı bulmaya çalışır.

İlk olarak, getAll() metodunu kullanarak tüm yazarları içeren bir liste elde edilir. Daha sonra, elde edilen bu liste üzerinde bir döngü başlatılır ve her bir yazar nesnesi üzerinde kontrol yapılır. Her bir yazar nesnesinin kimlik numarası (authorId) belirtilen authorId ile eşleşiyorsa, o yazar nesnesi döndürülür.

Eğer belirtilen kimlik numarasına sahip bir yazar bulunamazsa, metot null değeri döndürür ve 'Yazar bulunamadı' şeklinde bir açıklama belirtilir. Bu, belirtilen kimlik numarasına sahip yazarın listede olmadığı durumu ifade eder.

Bu metod, belirli bir yazar kimliğiyle ilişkilendirilmiş olan yazarı bulmak için kullanılan temel bir işlevselliği sağlar. Listede arama yaparak belirli bir yazarı bulur ve bulunan yazarı döndürür veya null değeri döndürerek belirtilen kimlik numarasına sahip bir yazarın bulunmadığını belirtir.

```
@Override
public void update(Author author) {
    List<Author> authors = getAll();
    for (int i = 0; i < authors.size(); i++) {
        if (authors.get(i).getAuthorId() == author.getAuthorId()) {
            authors.set(i, author);
            break;
        }
    }
    saveAllAuthors(authors);
}
```

Bu metod, mevcut bir yazarın bilgilerini güncellemek için kullanılır. İşlevselliği gereği, mevcut tüm yazarları içeren bir liste alır ve bu listede belirli bir yazarın kimliği (authorId) ile eşleşen yazarı bulmaya çalışır. Eşleşen yazar bulunduğunda, bu yazarın bilgileri, parametre olarak verilen yeni yazar bilgileri ile güncellenir.

İlk olarak, getAll() metodunu kullanarak tüm yazarları içeren bir liste elde edilir. Daha sonra, bu liste üzerinde bir döngü başlatılır ve her bir yazar nesnesinin kimlik numarası (authorId) belirtilen yazarın kimliği ile eşleşip eşleşmediği kontrol edilir. Eşleşen yazar bulunduğunda, yazarın bilgileri parametre olarak verilen yeni yazar bilgileri ile güncellenir ve döngü sonlandırılır.

Güncellenmiş yazar listesi, saveAllAuthors(authors) metoduna parametre olarak geçirilerek dosyaya kaydedilir. Bu sayede, güncellenmiş yazar bilgileri kalıcı olarak saklanır ve dosya üzerindeki değişiklikler uygulanır.

Bu metod, mevcut bir yazarın bilgilerini güncellemek için kullanılan temel bir işlevselliği sağlar. Listede güncelleme yaparak belirli bir yazarın bilgilerini günceller ve bu güncellemeleri dosyaya kaydederek kalıcı hale getirir.

```
@Override
public void delete(int authorId) {
    List<Author> authors = getAll();
    for (int i = 0; i < authors.size(); i++) {
        if (authors.get(i).getAuthorId() == authorId) {
            authors.remove(i);
            break;
        }
    }
    saveAllAuthors(authors);
}
```

Bu metod, belirtilen yazar kimliği (authorId) ile ilişkilendirilmiş olan yazarı listeden kaldırmak için kullanılır. İşlevselliği gereği, mevcut tüm yazarları içeren bir liste alır ve bu listede belirli bir yazarın kimliği ile eşleşen yazarı bulmaya çalışır. Eşleşen yazar bulunduğunda, bu yazar listeden kaldırılır.

İlk olarak, getAll() metodunu kullanarak tüm yazarları içeren bir liste elde edilir. Daha sonra, bu liste üzerinde bir döngü başlatılır ve her bir yazar nesnesinin kimlik numarası (authorId) belirtilen yazarın kimliği ile eşleşip eşleşmediği kontrol edilir. Eşleşen yazar bulunduğunda, bu yazar listeden kaldırılır ve döngü sonlandırılır.

Yazarın listeden kaldırılması işlemi tamamlandıktan sonra, güncellenmiş yazar listesi saveAllAuthors(authors) metoduna parametre olarak geçirilerek dosyaya kaydedilir. Bu sayede, güncellenmiş yazar listesi kalıcı olarak saklanır ve dosya üzerindeki değişiklikler uygulanır.

Bu metod, belirtilen yazar kimliği ile ilişkilendirilmiş olan yazarı listeden kaldırmak için kullanılan temel bir işlevselliği sağlar. Liste üzerinde değişiklik yaparak belirli bir yazarı listeden kaldırır ve bu değişiklikleri dosyaya kaydederek kalıcı hale getirir.

Aynı şekilde diğer sınıflar içinde CRUD işlemler dosya işlemleri üzerinden gerçekleştirilmiştir. Projenin gelecekte gerçek bir veri tabanı ile kullanılmak istendiğinde JpaRepository üzerinde gerçekleştirilmiştir.

JpaRepository, Spring Framework ile birlikte kullanılan bir arayüzdür ve JPA (Java Persistence API) ile etkileşimde bulunmak için kullanılır. JPA, Java ortamında nesne ilişkisel eşleme (Object-Relational Mapping - ORM) için standart bir API sağlar. JpaRepository, bu API'yi kullanarak veritabanı işlemlerini gerçekleştirmeyi kolaylaştırır. JpaRepository, genellikle Spring Data JPA modülü altında bulunur ve JpaRepository arayüzünün bir alt kümesidir. JpaRepository, JpaRepository arayüzünü genişletir ve JPA tarafından sağlanan temel veritabanı işlemlerini (kaydetme, güncelleme, silme, sorgulama) gerçekleştirmek için önceden tanımlanmış yöntemleri içerir. JpaRepository'nin temel avantajlarından biri,

veritabanı işlemlerini gerçekleştirmek için yazılması gereken tekrarlayan kod miktarını azaltmasıdır. Bu arayüz, JpaRepository'nin sağladığı yöntemleri kullanarak veritabanı işlemlerini otomatik olarak gerçekleştirir. Ayrıca, JpaRepository'nin kullanımı, kodun daha temiz ve okunabilir olmasını sağlar. Özetlemek gerekirse, JpaRepository, Spring Framework kullanılarak JPA ile etkileşimde bulunmak için kullanılan bir arayüzdür. Bu arayüz, JpaRepository arayüzünün bir alt kümesidir ve JPA tarafından sağlanan temel veritabanı işlemlerini gerçekleştirmek için önceden tanımlanmış yöntemleri içerir. Bu, veritabanı işlemlerini gerçekleştirmeyi kolaylaştırır ve tekrarlayan kod miktarını azaltır.

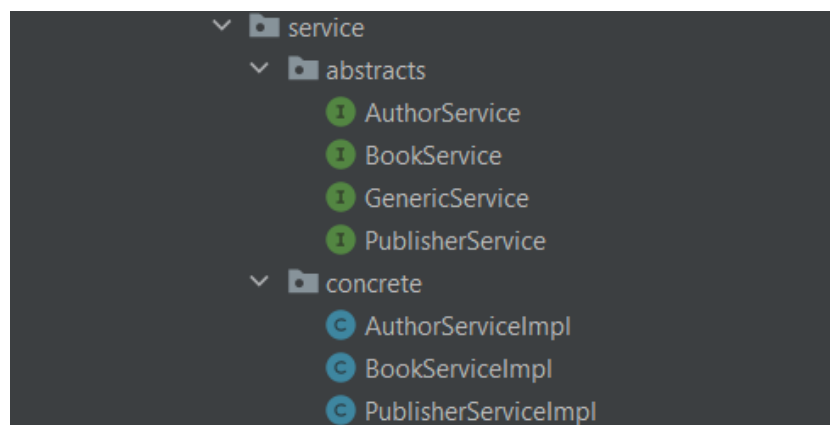
```
@Repository
public interface AuthorRepository extends JpaRepository<Author,Integer> {
    no usages
    | boolean existsByName(String name);
}
```

```
@Repository
public interface BookRepository extends JpaRepository<Book,Integer> {
    no usages
    boolean existByName(String name);
}
```

```
@Repository
public interface PublisherRepository extends JpaRepository<Publisher,Integer> {
    no usages
    boolean existByName(String name);
}
```

Gerçek veri tabanına geçilmek istenildiğinde servis sınıflarında JpaRepository instanceları kullanılması yeterli olacaktır. Projede bağımlılığı en aza indirenmiştir servislerin birbiri ile iletişimi tamamen interfacer aracılığı ile gerçekleştirilmiştir.

Servis Sınıfları



Servis sınıfları, bir yazılım uygulamasının iş mantığını gerçekleştiren ve iş süreçlerini yöneten bileşenlerdir. Spring framework'ünde servis sınıfları genellikle @Service annotation'ı ile işaretlenirler ve uygulamanın iş mantığının birleştirildiği yer olarak kullanılırlar.

Servis sınıfları, genellikle uygulamanın karmaşık işlemlerini gerçekleştirir. Bu işlemler arasında veritabanı işlemleri, dış sistemlerle iletişim, iş kurallarının uygulanması gibi çeşitli görevler bulunabilir. Servis sınıfları, bu işlemleri gerçekleştirirken kontrolcülerden (controllers) ve veritabanı erişim katmanlarından (repositories) gelen istekleri kabul eder ve işler.

Spring framework'ünde servis sınıflarının kullanılması, iş mantığının diğer bileşenlerden ayrılmasını sağlar. Bu, uygulamanın daha modüler hale gelmesini ve daha kolay bakım yapılabilir olmasını sağlar. Ayrıca, servis sınıfları genellikle transaksyon yönetimi gibi ortak işlevleri de sağlarlar.

Servis sınıfları ayrıca, iş mantığının tek bir yerde toplanmasını sağlar, böylece kod tekrarını azaltır ve daha temiz bir kod tabanı oluşturur. Bu, uygulamanın daha kolay test edilmesini ve geliştirilmesini sağlar.

Özetlemek gerekirse, servis sınıfları, bir yazılım uygulamasının iş mantığını gerçekleştiren ve iş süreçlerini yöneten bileşenlerdir. Spring framework'ünde servis sınıfları, uygulamanın modülerliğini artırır, kod tekrarını azaltır ve daha temiz bir kod tabanı oluşturur. Projemizde ekstra olarak servisleri generic yapılarla kullanarak daha modüler ve sürdürülebilir bir kod elde etmiş olduk.

```

public interface GenericService<T> {
    3 implementations
    List<T> getAll();
    3 implementations
    void save(T object);
    3 implementations
    T getById(int id);
    3 implementations
    void update(T object);
    3 implementations
    void delete(int id);
}

```

```

6 usages 1 implementation
public interface AuthorService extends GenericService<Author> {
}

```

```

6 usages 1 implementation
public interface BookService extends GenericService<Book> {
}

```

```

public interface PublisherService extends GenericService<Publisher> {
    1 usage 1 implementation
    List<String> listBooksAndAuthorsOfTwoPublishers();
}

```

Yazılım geliştirme süreçlerinde tekrar kullanılabilirlik, kod kalitesi ve esneklik önemli faktörlerdir. Bu hedeflere ulaşmada Java'da generic yapıların kullanımı büyük önem taşır. servis katmanlarında generic yapıların nasıl kullanılabileceğini ve bu kullanımının avantajlarını ele alacağız.

Servis katmanı, iş mantığının uygulandığı ve dış dünya ile iletişimin sağlandığı bir bileşendir. Bu katmanda, genellikle CRUD işlemleri (Create, Read, Update, Delete) gibi temel veritabanı işlemleri gerçekleştirilir. Ancak, farklı varlık türleri için aynı işlemleri tekrar tekrar yazmak, kod tekrarına ve karmaşıklığa neden olabilir.

Bu durumu önlemek için generic yapılar devreye girer. Generic yapılar, farklı türde veri türleri üzerinde çalışabilen ve kod tekrarını azaltan bir mekanizmadır. Servis katmanlarında generic yapıların kullanılması, aynı CRUD işlemlerinin farklı varlık türleri için tek bir yerden

yönetilmesini sağlar. Bu da kodun daha temiz, daha modüler ve daha okunabilir olmasını sağlar.

```
package com.example.JavaTask.service.concrete;

import java.util.List;

import com.example.JavaTask.model.Author;
import com.example.JavaTask.repository.abstracts.FileAuthorRepository;
import com.example.JavaTask.service.abstracts.AuthorService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

2 usages
@Service
public class AuthorServiceImpl implements AuthorService {

    5 usages
    @Autowired
    private FileAuthorRepository authorRepository;

    @Override
    public List<Author> getAll() { return authorRepository.getAll(); }
    @Override
    public void save(Author author) { authorRepository.save(author); }
    @Override
    public Author getById(int authorId) { return authorRepository.getById(authorId); }
    @Override
    public void update(Author author) { authorRepository.update(author); }
    @Override
    public void delete(int authorId) { authorRepository.delete(authorId); }

}
```

AuthorServiceImpl sınıfı, uygulamamızdaki yazar işlemlerini gerçekleştiren servis sınıfını temsil eder. Bu sınıf, AuthorService arayüzünü uygular ve bu arayüzde tanımlanan metodları kullanarak yazar işlemlerini yönetir. Spring framework'ünde @Service annotation'ı ile işaretlenmiş olan bu sınıf, Spring konteyneri tarafından yönetilir ve dependency injection ile FileAuthorRepository bağımlılığını çözer.

- getAll(): Bu metod, tüm yazarları almak için FileAuthorRepository sınıfındaki getAll() metodunu çağırır ve aldığı yazar listesini döndürür.
- save(): Yeni bir yazarı kaydetmek için kullanılır. Parametre olarak aldığı yazar nesnesini FileAuthorRepository sınıfındaki save() metoduna ileterek kaydetme işlemini gerçekleştirir.

- `getById()`: Belirli bir yazarın bilgilerini almak için kullanılır. Parametre olarak aldığı yazar kimliği (`authorId`) ile `FileAuthorRepository` sınıfındaki `getById()` metodunu çağırarak ilgili yazarı alır ve geri döndürür.
- `update()`: Varolan bir yazarın bilgilerini güncellemek için kullanılır. Parametre olarak aldığı yazar nesnesini `FileAuthorRepository` sınıfındaki `update()` metoduna ileterek güncelleme işlemini gerçekleştirir.
- `delete()`: Belirli bir yazarı silmek için kullanılır. Parametre olarak aldığı yazar kimliği (`authorId`) ile `FileAuthorRepository` sınıfındaki `delete()` metodunu çağırarak ilgili yazarı siler.

Bu şekilde, `AuthorServiceImpl` sınıfı, uygulamamızdaki yazar işlemlerinin merkezi bir noktası haline gelir ve bu işlemleri yönetir. Bu da kodun daha modüler, bakımı daha kolay ve daha okunabilir olmasını sağlar. Aynı şekilde diğer servis sınıflarımızda instance olarak aldığı repository nesnesindeki metodu kullanarak CRUD işlemlerini gerçekleştirir.

Controller Sınıfı

Controller sınıfları, bir web uygulamasının HTTP isteklerini karşılayan ve işleyen bileşenlerdir. Kullanıcı tarafından tarayıcıya gönderilen istekler, Controller sınıflarına iletilir ve bu sınıflar isteğe uygun olarak işlem yaparlar. Controller sınıfları, gelen isteğe göre uygun iş mantığını çağırır, gerekli verileri alır veya işler ve sonuç olarak HTTP yanıtını oluşturur.

Spring Framework'te Controller sınıfları genellikle `@Controller` veya `@RestController` annotation'ı ile işaretlenirler. `@Controller` annotation'ı, HTML görünümüleri gibi geleneksel web uygulamaları için kullanılırken, `@RestController` annotation'ı, RESTful servisler gibi veri tabanı veya dış sistemlerle etkileşimde bulunan API'ler için kullanılır.

Controller sınıfları, Spring konteyneri tarafından yönetilir ve dependency injection ile diğer bileşenlere (servisler, repositoryler vb.) erişebilirler. Bu, Controller sınıflarının bağımlılıkları enjekte edilerek test edilebilir hale gelmesini sağlar.

Controller sınıfları, Spring Framework'te web uygulamalarında gelen istekleri işleyen ve uygun yanıtları oluşturan önemli bileşenlerdir. Modülerlik, ayırma, HTTP isteklerini yönetme, API sunma ve test edilebilirlik gibi avantajlar sağlarlar. Bu avantajlar, uygulamanın geliştirilmesini ve bakımını kolaylaştırır, kodun daha temiz ve okunabilir olmasını sağlar.

Controller sınıfları, iş mantığını diğer katmanlardan (servisler, repositoryler vb.) ayırarak uygulamanın modüler olmasını sağlar. Bu, uygulamanın bakımını kolaylaştırır ve kodun daha okunabilir olmasını sağlar.

Controller sınıfları, HTTP isteklerini yöneterek gelen isteğe uygun işlemleri gerçekleştirir. Bu, uygulamanın dış dünya ile etkileşimini sağlar ve kullanıcıların isteklerine yanıt verir.

`@RestController` annotation'ı ile işaretlenen Controller sınıfları, RESTful API'lerin oluşturulmasını sağlar. Bu, uygulamanın dış dünya ile kolayca iletişim kurmasını ve veri paylaşımını sağlar.

```

@RestController
@RequestMapping("/authors")
public class AuthorController {

    5 usages
    @Autowired
    private AuthorService authorServiceImpl;

    no usages
    @GetMapping("/")
    public List<Author> getAllAuthors() { return authorServiceImpl.getAll(); }

    no usages
    @PostMapping("/")
    public void addAuthor(@RequestBody Author author) { authorServiceImpl.save(author); }

    no usages
    @GetMapping("/{authorId}")
    public Author getAuthorById(@PathVariable int authorId) { return authorServiceImpl.getById(authorId); }

    no usages
    @PutMapping("/{authorId}")
    public void updateAuthor(@PathVariable int authorId, @RequestBody Author author) {
        author.setAuthorId(authorId);
        authorServiceImpl.update(author);
    }

    no usages
    @DeleteMapping("/{authorId}")
    public void deleteAuthor(@PathVariable int authorId) { authorServiceImpl.delete(authorId); }
}

```

AuthorController sınıfı, uygulamamızın yazar işlemlerini yöneten Controller bileşenini temsil eder. Bu sınıf, Spring Framework'te @RestController annotation'ı ile işaretlenmiş ve "/authors" endpoint'ini kullanarak HTTP isteklerini karşılar.

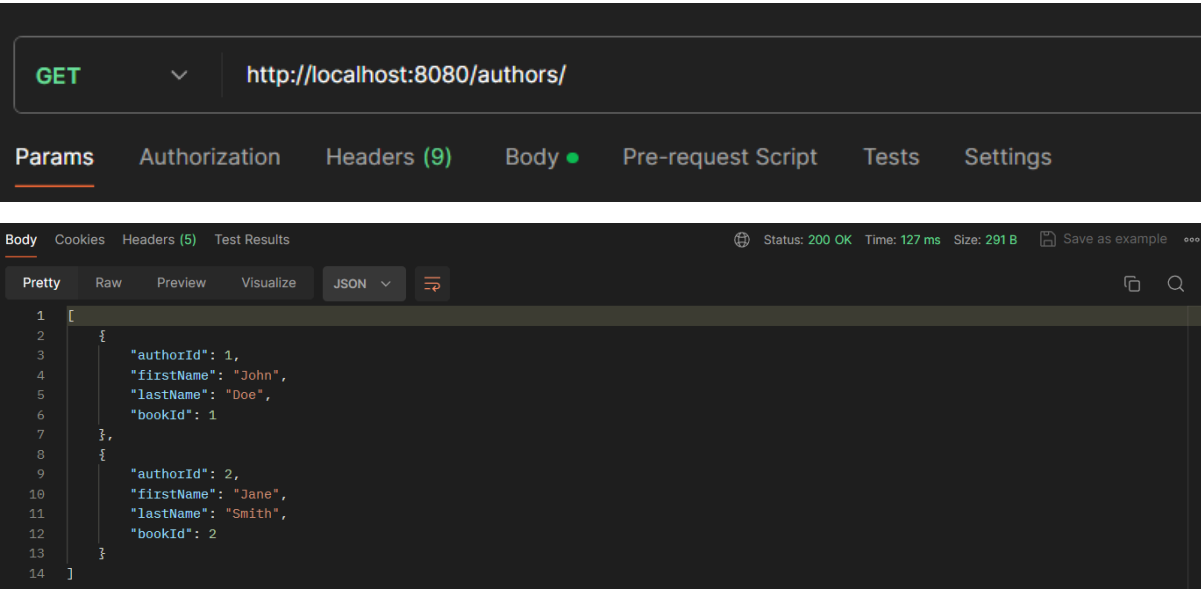
- getAllAuthors(): Bu metod, tüm yazarları almak için HTTP GET isteğini işler. AuthorService arayüzünde tanımlı olan getAll() metodunu çağırarak tüm yazarları alır ve bu yazar listesini döndürür.
- addAuthor(): Yeni bir yazar eklemek için HTTP POST isteğini işler. Gelen HTTP isteğinin gövdesinde bulunan yazarı alır ve AuthorService arayüzündeki save() metodunu çağırarak yazarı kaydeder.
- getAuthorById(): Belirli bir yazarı almak için HTTP GET isteğini işler. Path parametresinde belirtilen yazar kimliğini alır ve AuthorService arayüzündeki getById() metodunu çağırarak ilgili yazarı alır ve geri döndürür.
- updateAuthor(): Varolan bir yazarın bilgilerini güncellemek için HTTP PUT isteğini işler. Path parametresinde belirtilen yazar kimliği ve gelen HTTP isteğinin gövdesinde bulunan güncellenmiş yazar bilgilerini alır, AuthorService arayüzündeki update() metodunu çağırarak yazarı günceller.
- deleteAuthor(): Belirli bir yazarı silmek için HTTP DELETE isteğini işler. Path parametresinde belirtilen yazar kimliğini alır ve AuthorService arayüzündeki delete() metodunu çağırarak ilgili yazarı siler.

AuthorController sınıfı, gelen HTTP isteklerini alır ve ilgili işlemleri gerçekleştirmek için AuthorService bileşenini kullanır. Bu şekilde, uygulamamızın farklı katmanları arasında iletişim sağlar ve iş mantığının doğru bir şekilde çalışmasını sağlar. AuthorController sınıfı, Spring Framework'te HTTP isteklerini alıp işleyen ve uygun yanıtları oluşturan bir bileşendir. HTTP GET, POST, PUT ve DELETE isteklerini işleyerek uygulamamızın yazar işlemlerini yönetir. Bu, uygulamanın dış dünya ile etkileşimini sağlar ve kullanıcıların isteklerine yanıt verir.

Postman Sonuçları

Author modelinin Yazar.txt dosyasındaki bütün kayıtları getirir.

```
@GetMapping("/")
public List<Author> getAllAuthors() { return authorServiceImpl.getAll(); }
```

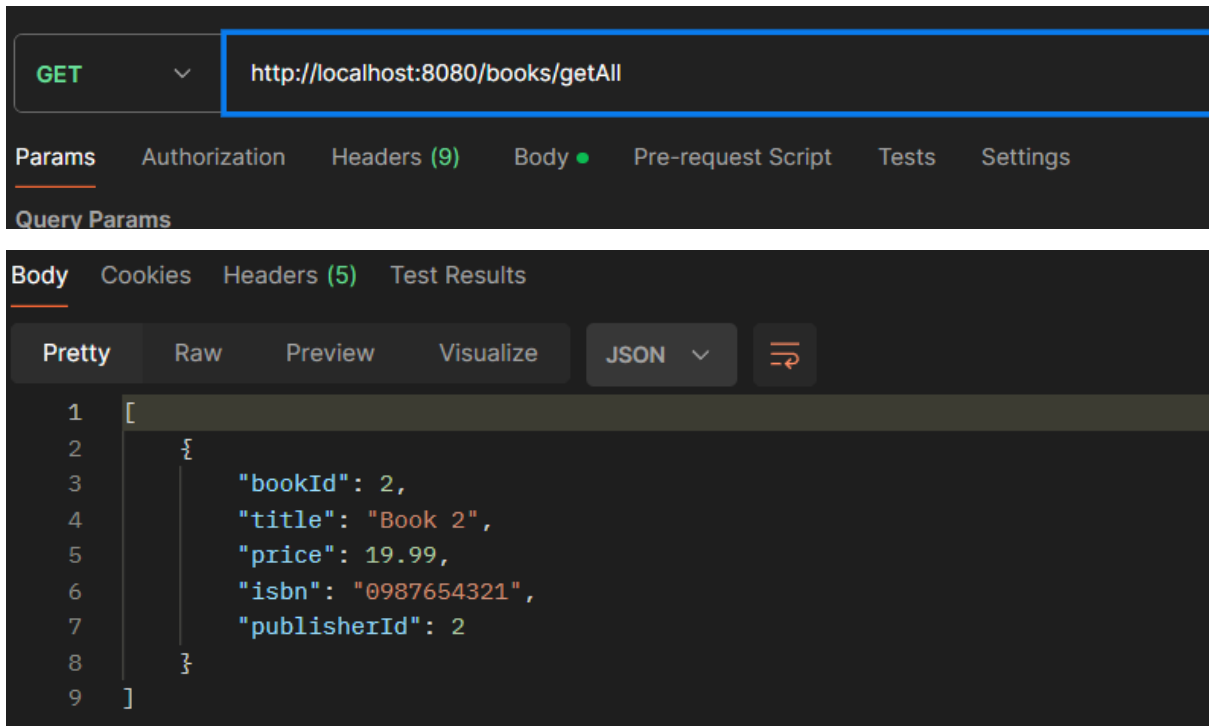


The screenshot shows a Postman interface with a GET request to `http://localhost:8080/authors/`. The response is a JSON array of two author objects. The status is 200 OK, time is 127 ms, and size is 291 B.

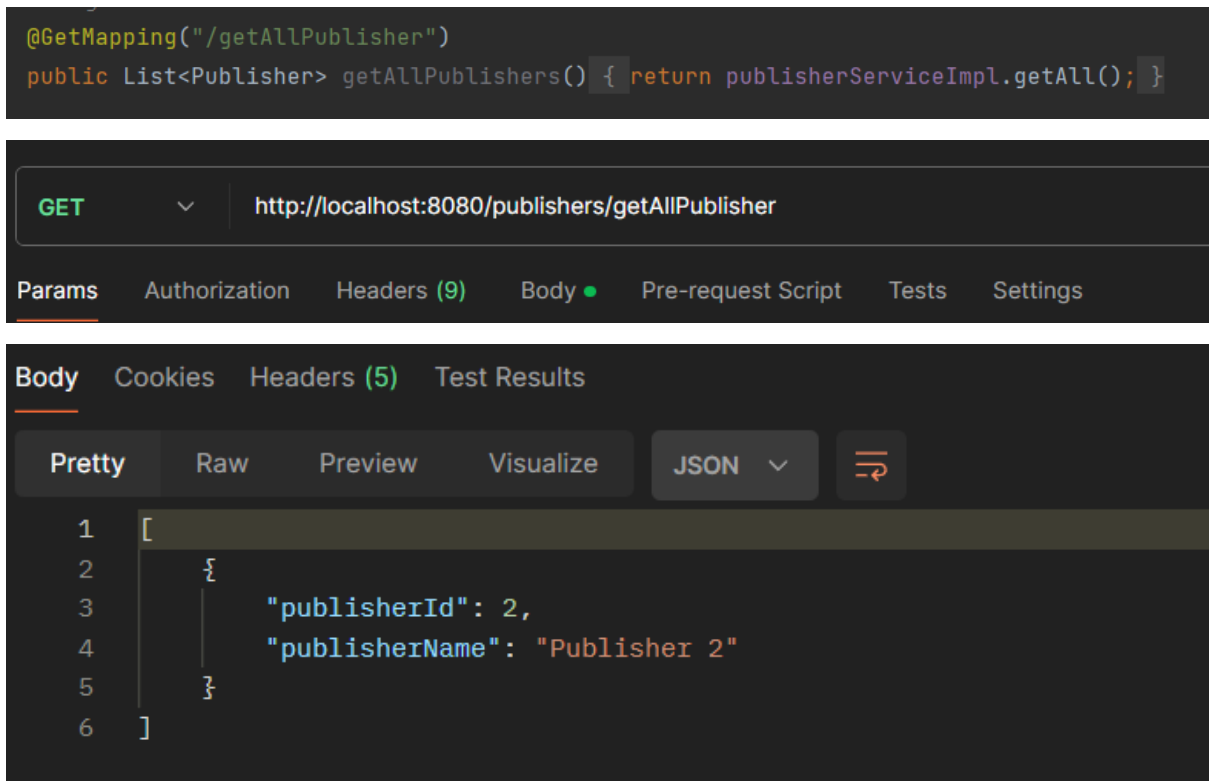
```
[
  {
    "authorId": 1,
    "firstName": "John",
    "lastName": "Doe",
    "bookId": 1
  },
  {
    "authorId": 2,
    "firstName": "Jane",
    "lastName": "Smith",
    "bookId": 2
  }
]
```

Book modelinin Kitap.txt dosyasındaki bütün kayıtları getirir.

```
@GetMapping("/getAll")
public List<Book> getAllBooks() {
    return bookServiceImp.getAll();
}
```



Publisher modelinin Yayınevi.txt dosyasındaki bütün kayıtları getirir.



Proje gereksinim dosyasında özellikle istenen Tüm kitapları ve ilgili yazarlarıyla birlikte 2 yayınevi listele metodu.

```
@GetMapping("/listPublisherWithBookandAuthor")
public List<String> listBooksAndAuthorsOfTwoPublishers(){
    return publisherServiceImpl.listBooksAndAuthorsOfTwoPublishers();
}
```

GET http://localhost:8080/publishers/listPublisherWithBookandAuthor

Params Authorization Headers (9) Body • Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results

Status: 200 OK Time: 9 ms Size: 229 B

Pretty Raw Preview Visualize JSON

```
1 [
2   "Yayınevi: Publisher 2",
3   "Kitap: Book 2",
4   "Yazar: Jane Smith",
5   "",
6   ""
]
```

Book modelinin Kitap.txt dosyasına yeni kayıt ekler.

```
@PostMapping("/")
public void addBook(@RequestBody Book book) { bookServiceImpl.save(book); }
```

POST http://localhost:8080/books/

Params Authorization Headers (9) Body • Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "bookId": 3,
3   "title": "İnsan ne ile yaşar",
4   "price": 29.99,
5   "isbn": "978-3-16-148410-0",
6   "publisherId": 123
7 }
```

Body Cookies Headers (4) Test Results

Status: 200 OK Time: 51 ms Size: 123 B

Pretty Raw Preview Visualize Text

```
1 
```

```
2;Book 2;19.99;0987654321;2
3;İnsan ne ile yaşar;29.99;978-3-16-148410-0;123
```

Test Sınıfları

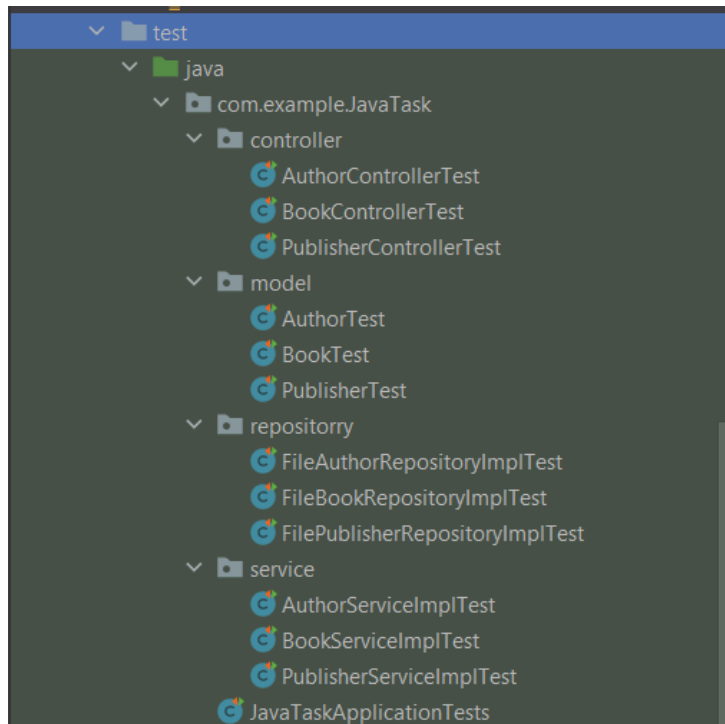
JUnit, Java tabanlı bir birim test çerçevesidir ve yazılımın belirli birimlerinin doğru çalışıp çalışmadığını kontrol etmek için kullanılır. Unit testler, yazılımın en küçük işlevsel birimlerini (metodlar, fonksiyonlar, sınıflar) test etmek için kullanılır. Birim testler, bu küçük birimlerin beklenen girdilerle doğru çıktıları üretip üretmediğini belirler.

Spring Framework altında yazılım geliştirmenin avantajlarından biri, Spring tarafından sağlanan özellikler sayesinde birim testlerin kolayca yazılabilmesidir. Spring'in IOC (Inversion of Control) ve DI (Dependency Injection) prensipleri, kodun bağımlılıklarını yönetir ve bu da birim testlerin yazılmasını kolaylaştırır. Örneğin, bağımlılıkların enjekte edilmesi, mock nesnelerin kullanılması ve Spring Boot'un sunduğu test ortamları gibi özellikler, testlerin yazılmasını ve çalıştırılmasını kolaylaştırır.

Unit testlerin avantajları şunlardır:

- **Kodun Doğruluğunu Sağlar:** Unit testler, yazılan kodun beklenen sonuçları üretip üretmediğini kontrol eder. Bu sayede kodun doğruluğu sağlanır ve hata bulunması kolaylaşır.
- **Güvenilirliği Artırır:** Doğru bir şekilde yazılmış ve sürekli çalıştırılan unit testler, kodun güvenilirliğini artırır. Kod değişiklikleri sonrasında testlerin başarılı olması, değişikliklerin beklenen sonuçları etkilemediğini gösterir.
- **Dokümantasyon Gibi Olur:** Unit testler, yazılımın belirli birimlerinin nasıl çalışması gerektiğini açıkça tanımlar. Bu nedenle, kodun anlaşılmasını ve bakımını kolaylaştırır.
- **Refactoringi Kolaylaştırır:** Kodun yeniden düzenlenmesi ve geliştirilmesi (refactoring) sırasında, varolan unit testlerin başarısız olması durumunda hemen fark edilir. Bu da refactoring işlemlerinin güvenli bir şekilde yapılmasını sağlar.
- **Hata Ayıklamayı Kolaylaştırır:** Unit testler, hataların nerede olduğunu hızlı bir şekilde tespit etmeyi sağlar. Hatalı birimlerin belirlenmesi ve izole edilmesi, hata ayıklama sürecini hızlandırır.
- **Yazılımın Sürekli İyileştirilmesini Sağlar:** Sürekli olarak çalışan unit testler, yazılımın sürekli olarak iyileştirilmesini ve bakımını sağlar. Bu da yazılımın kalitesini artırır ve müşteri memnuniyetini sağlar.

Sonuç olarak, unit testler, yazılım geliştirme sürecinde önemli bir yer tutar ve kodun doğruluğunu, güvenilirliğini ve kalitesini artırır. Bu nedenle, yazılım geliştiricilerin test odaklı bir yaklaşım benimsemeleri ve yazdıkları kodu düzenli olarak test etmeleri önemlidir.



```
import static org.mockito.Mockito.*;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBui
lders.*;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatch
ers.*;

import java.util.Arrays;
import java.util.List;

import com.example.JavaTask.model.Author;
import com.example.JavaTask.service.abstracts.AuthorService;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoCo
nfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
```

```

@SpringBootTest
@AutoConfigureMockMvc
public class AuthorControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private AuthorService authorService;

    private Author author1;
    private Author author2;

    @BeforeEach
    public void setUp() {
        author1 = new Author(1, "John", "Doe", 1);
        author2 = new Author(2, "Jane", "Smith", 2);
    }

    @Test
    public void testGetAllAuthors() throws Exception {
        List<Author> authors = Arrays.asList(author1, author2);
        when(authorService.getAll()).thenReturn(authors);

        mockMvc.perform(get("/authors/"))
            .andExpect(status().isOk())

            .andExpect(content().contentType(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$.authorId").value(1))

            .andExpect(jsonPath("$.firstName").value("John"))

            .andExpect(jsonPath("$.lastName").value("Doe"))
            .andExpect(jsonPath("$.bookId").value(1))
            .andExpect(jsonPath("$.authorId").value(2))

            .andExpect(jsonPath("$.firstName").value("Jane"))

            .andExpect(jsonPath("$.lastName").value("Smith"))
            .andExpect(jsonPath("$.bookId").value(2));
    }

    @Test
    public void testAddAuthor() throws Exception {

```

```

        mockMvc.perform(post("/authors/"))

        .contentType(MediaType.APPLICATION_JSON)

        .content("{ \"authorId\": 3,
\"firstName\": \"Alice\", \"lastName\": \"Brown\", \"bookId\":
3 }"))

        .andExpect(status().isOk());

        verify(authorService,
times(1)).save(any(Author.class));
    }

    @Test
    public void testGetAuthorById() throws Exception {
        when(authorService.getById(1)).thenReturn(author1);

        mockMvc.perform(get("/authors/1"))

        .andExpect(status().isOk())

        .andExpect(content().contentType(MediaType.APPLICATION_JSON))

        .andExpect(jsonPath("$.authorId").value(1))

        .andExpect(jsonPath("$.firstName").value("John"))

        .andExpect(jsonPath("$.lastName").value("Doe"))

        .andExpect(jsonPath("$.bookId").value(1));
    }

    @Test
    public void testUpdateAuthor() throws Exception {
        mockMvc.perform(put("/authors/1")

        .contentType(MediaType.APPLICATION_JSON)

        .content("{ \"authorId\": 1,
\"firstName\": \"Updated John\", \"lastName\": \"Updated
Doe\", \"bookId\": 1 }"))

        .andExpect(status().isOk());

        verify(authorService,
times(1)).update(any(Author.class));
    }

    @Test
    public void testDeleteAuthor() throws Exception {
        mockMvc.perform(delete("/authors/1"))

```

```
        .andExpect(status().isOk());

        verify(authorService, times(1)).delete(1);
    }
}
```

AuthorControllerTest sınıfı, AuthorController sınıfının birim testlerini gerçekleştiren bir JUnit test sınıfıdır. Bu sınıf, Spring Framework altında çalışan AuthorController sınıfının tüm metodlarını test eder.

Test Metodları ve İşlevleri

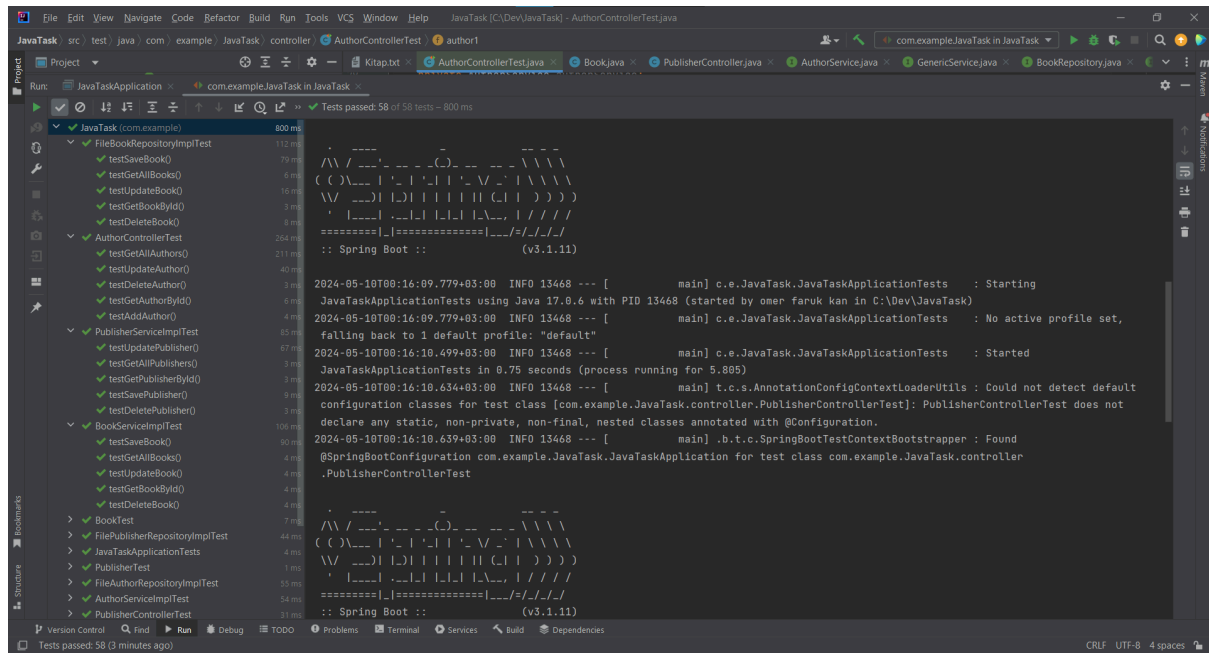
- testGetAllAuthors(): Bu metod, AuthorController sınıfındaki getAllAuthors() metodunu test eder. MockMvc kullanarak HTTP GET isteği yapar ve beklenen sonuçların dönüp dönmediğini kontrol eder.
- testAddAuthor(): Yeni bir yazar eklemek için addAuthor() metodunu test eder. MockMvc kullanarak HTTP POST isteği yapar ve bu isteğin başarılı bir şekilde tamamlanıp tamamlanmadığını kontrol eder.
- testGetAuthorById(): Belirli bir yazarı almak için getAuthorById() metodunu test eder. MockMvc kullanarak HTTP GET isteği yapar ve beklenen sonuçların dönüp dönmediğini kontrol eder.
- testUpdateAuthor(): Varolan bir yazarı güncellemek için updateAuthor() metodunu test eder. MockMvc kullanarak HTTP PUT isteği yapar ve bu isteğin başarılı bir şekilde tamamlanıp tamamlanmadığını kontrol eder.
- testDeleteAuthor(): Belirli bir yazarı silmek için deleteAuthor() metodunu test eder. MockMvc kullanarak HTTP DELETE isteği yapar ve bu isteğin başarılı bir şekilde tamamlanıp tamamlanmadığını kontrol eder.

MockMvc Kullanımı

Bu test sınıfı, Spring Framework tarafından sağlanan MockMvc kütüphanesini kullanarak HTTP istekleri yapar ve bu isteklerin sonuçlarını kontrol eder. @AutoConfigureMockMvc annotation'ı sayesinde MockMvc otomatik olarak yapılandırılır ve testler yazılırken gerçek HTTP isteklerini simüle eder.

MockBean Kullanımı

@MockBean annotation'ı, AuthorService bileşenini bir mock nesnesi olarak enjekte eder. Bu sayede, AuthorController sınıfı AuthorService bileşenine bağımlı olduğunda, gerçek bir AuthorService yerine mock AuthorService kullanılır. Bu da testlerin bağımsız bir şekilde çalışmasını sağlar.



Uygulamam Junit test çerçevesinde test sınıflarının kodlarında yazmış bulunmaktayım ve bütün testlerden başarılı bir şekilde geçmiştir. Başarılı birim testleri, yazılımın beklenen sonuçları ürettiğini ve doğru çalıştığını gösterir. Bu da uygulamanın güvenilirliğini artırır ve kullanıcıların güvenini kazanır. Birim testleri, kodun kalitesini artırır. Doğru bir şekilde yazılmış ve sürekli çalıştırılan testler, kodun hatalı veya beklenmeyen davranışlar sergilemesini engeller. Birim testleri, hataların erken tespit edilmesini sağlar. Kod değişiklikleri sonrasında birim testlerin başarısız olması durumunda, hata hızlı bir şekilde tespit edilir ve düzeltilir. Başarılı birim testleri, kodun daha kolay bakımını sağlar. Yapılan değişikliklerin mevcut işlevselliği etkileyip etkilemediği hızlı bir şekilde belirlenir ve uygun şekilde işlenir. Birim testler, kodun nasıl kullanılması gerektiği ve beklenen davranışlar hakkında değerli bilgiler sağlar. Bu da kodun anlaşılmasını ve yeni geliştiricilerin projeye katılmasını kolaylaştırır.

Sürekli Entegrasyon ve Dağıtım (CI/CD): Başarılı birim testleri, sürekli entegrasyon ve dağıtım süreçlerini destekler. CI/CD ortamlarında, her kod değişikliğinden sonra birim testler otomatik olarak çalıştırılır ve hata durumunda bildirimler gönderilir. Bu sayede, hataların erken tespiti ve hızlı düzeltilmesi sağlanır.

Run: JavaTaskApplication x com.example.JavaTask i	
✓ JavaTask (com.example)	800 ms
✓ FileBookRepositoryImplTest	112 ms
✓ testSaveBook()	79 ms
✓ testGetAllBooks()	6 ms
✓ testUpdateBook()	16 ms
✓ testGetBookById()	3 ms
✓ testDeleteBook()	8 ms
✓ AuthorControllerTest	264 ms
✓ testGetAllAuthors()	211 ms
✓ testUpdateAuthor()	40 ms
✓ testDeleteAuthor()	3 ms
✓ testGetAuthorById()	6 ms
✓ testAddAuthor()	4 ms
✓ PublisherServiceImplTest	85 ms
✓ testUpdatePublisher()	67 ms
✓ testGetAllPublishers()	3 ms
✓ testGetPublisherById()	3 ms
✓ testSavePublisher()	9 ms
✓ testDeletePublisher()	3 ms
✓ BookServiceImplTest	106 ms
✓ testSaveBook()	90 ms
✓ testGetAllBooks()	4 ms
✓ testUpdateBook()	4 ms
✓ testGetBookById()	4 ms
✓ testDeleteBook()	4 ms
> ✓ BookTest	7 ms
> ✓ FilePublisherRepositoryImplTest	44 ms
> ✓ JavaTaskApplicationTests	4 ms
> ✓ PublisherTest	1 ms
> ✓ FileAuthorRepositoryImplTest	55 ms
> ✓ AuthorServiceImplTest	54 ms
> ✓ PublisherControllerTest	31 ms