Spring Framework’ünde Design Patternler

Design Patternler, yazılım geliştirmenin önemli bir parçasıdır. Bu çözümler yalnızca yinelenen sorunları çözmekle kalmaz, aynı zamanda geliştiricilerin ortak kalıpları tanıyarak bir framework’ün tasarımını anlamalarına yardımcı olur.

Bu makalede, Spring Framework'te kullanılan en yaygın dört design pattern’e bakacağız:

1. Singleton pattern
2. Factory Method pattern
3. Proxy pattern
4. Template pattern

Ayrıca Spring'in geliştiricilerin üzerindeki yükü azaltmak ve kullanıcıların sıkıcı görevleri hızla gerçekleştirmelerine yardımcı olmak için bu kalıpları nasıl kullandığına da bakacağız.

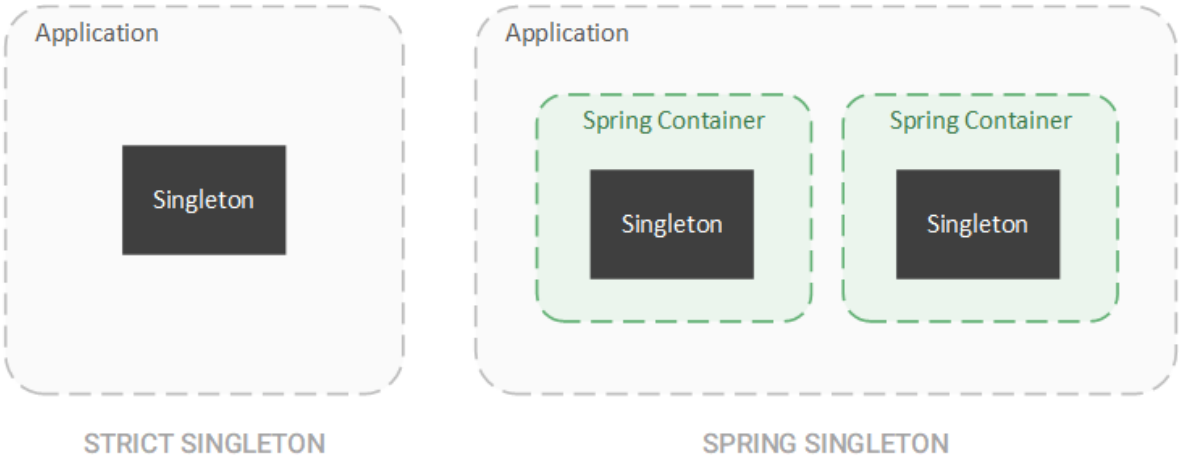
1. Singleton Pattern

Singleton deseni , uygulama başına bir nesnenin yalnızca bir örneğinin var olmasını sağlayan bir mekanizmadır. Bu model, paylaşılan kaynakları yönetirken veya günlüğe kaydetme gibi kesişen hizmetler sağlarken faydalı olabilir.

1.1 Singleton Beans

Genel olarak, bir singleton bir uygulama için küresel olarak benzersizdir, ancak Spring’de bu kısıtlama gevşetilir. Bunun yerine Spring, bir singleton'u Spring IoC konteyneri başına bir nesneyle kısıtlar. Pratikte bu, Spring'in uygulama contexti başına her tür için yalnızca bir bean oluşturacağı anlamına gelir.

Spring'in yaklaşımı, bir uygulamanın birden fazla Spring konteynerine sahip olabileceğinden, katı bir singleton tanımından farklıdır. Bu nedenle, birden fazla konteynırımız varsa, aynı sınıftan birden çok nesne tek bir uygulamada bulunabilir.



Varsayılan olarak Spring, tüm beanleri singleton olarak oluşturur.

1.2 Autowired Singletons

Örnek olarak bir uygulama context'ine 2 adet controller oluşturup bunlara aynı tipteki bir bean inject edilebilir.

İlk olarak Kitap domain objelerimizi yönetecek BookRepository oluşturuyoruz.

Sonra LibraryController oluşturuyoruz. Bu controller BookRepository'i kullanarak bir kütüphanedeki kitap sayısını return edecektir.

```
@RestController
public class LibraryController {

  @Autowired
  private BookRepository repository;

  @GetMapping("/count")
  public Long findCount() {
    System.out.println(repository);
    return repository.count();
  }
}
```

Son olarak BookController oluşturuyoruz. Bu controller da kitapla ilgili aksiyonları handle edebilmek var örneğin kitabın id'sine göre kitap bilgileri bulabilmek için.

```
@RestController
public class BookController {

  @Autowired
  private BookRepository repository;

  @GetMapping("/book/{id}")
  public Book findById(@PathVariable long id) {
    System.out.println(repository);
    return repository.findById(id).get();
  }
}
```

Bu uygulamayı başlattığımızda ve /count ve /book/1 endpointlere Get requesti gönderdiğimizde uygulamadaki loglarda BookRepository objesinin aynı id'ye sahip olduğunu görebiliriz.

BookRepository objesinin id'leri LibraryController ve BookController'da aynıdır. Bu da Spring'in aynı bean'i iki controller'a inject ettiğini kanıtlıyor.

BookRepository'nin farklı instance'larını bean scope'unu singleton'dan prototype'a çevirerek yaratabilirdik. Bunun için;
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE) annotation'ını kullanabiliriz.

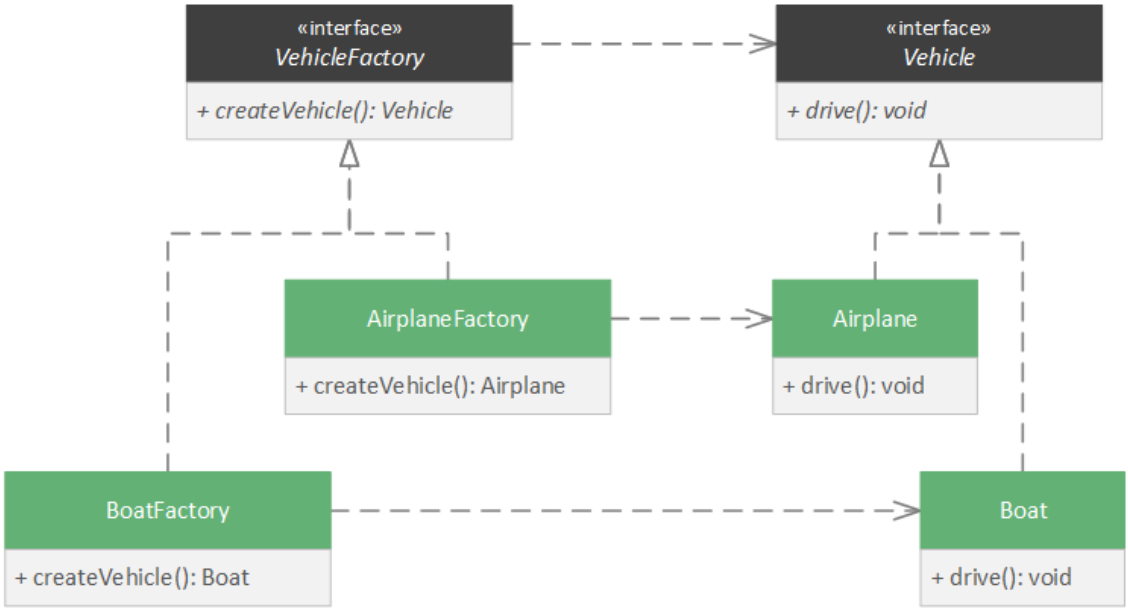
Eğer bunu yaparsak Spring oluşturduğu her bookRepository beani için farklı objeler yaratacaktırç Eğer controller'daki bookRepository'lerin id'lerine tekrar bakarsak id'lerinin aynı olmadığını görebiliriz.

2. Factory Method Pattern

Factory metod patterni istenen nesneyi oluşturmak için soyut bir yöntemle bir factory class'ını gerektirir.

Bazen belli context'lere göre farklı objeler oluşturmak isteyebiliyoruz.

Örek olarak bizim uygulamamız Vehicle objesine ihtiyaç duyuyor. Deniz ortamında Boat yaratmak istiyoruz ama hava ortamına Airplane yaratmak istiyoruz.



Buna ulaşmak için arzu edilen her obje için bir factory implementasyonu oluşturulabilir ve factory metodunda nesne return edilir.

2.1 Application Context

Spring bu tekniği Dependency Injection frameworkündün en tepesinde kullanır. Temel olarak Spring bean container'ına bean üreten bir factory gibi davranır. Böylece Spring BeanFactory interface'ini bir bean container'ının soyutlaması olarak

tanımlar.

```
public interface BeanFactory {

   getBean(Class<T> requiredType);
   getBean(Class<T> requiredType, Object... args);
   getBean(String name);

   // ...
]
```

Her getBean metodu, metoda sağlanan kriterlere (bean tipi, isim) göre bean return eden bir factory methodu olarak kabul edilir.

Spring sonrasında BeanFactory'i ApplicationContext interface'ini genişletir ve ekstra uygulama konfigürasyonları sağlar. Spring bu konfigürayonları bazı dış konfigürasyonlara göre bean container'larını başlatmak için kullanır örneğin xml dosyaları ve anotasyonlar.

AnnotationConfigApplicationContext'i ApplicationContext'in implementasyonu olarak kullanarak BeanFactory interface'inden türemiş değişik factory metodlarını kullanıp bean'ler oluşturabiliriz.

Önce basit bir uygulama konfigürasyonu oluşturuyoruz.

```
@Configuration
@ComponentScan(basePackageClasses = ApplicationConfig.class)
public class ApplicationConfig {
}
```

Sonrasında basit bir constructer'ı olmayan bir Foo class'ı oluşturuyoruz.

```
@Component
public class Foo {
}
```

Sonrasında bir constructor argümanı olan bir Bar class'ı oluşturuyoruz.

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class Bar {

   private String name;

   public Bar(String name) {
      this.name = name;
   }
```

```
   // Getter ...}
```

Son olarak ApplicationContext'in implementasyonu olan AnnotationConfigApplicationContext ile beanlerimizi oluşturuyoruz.

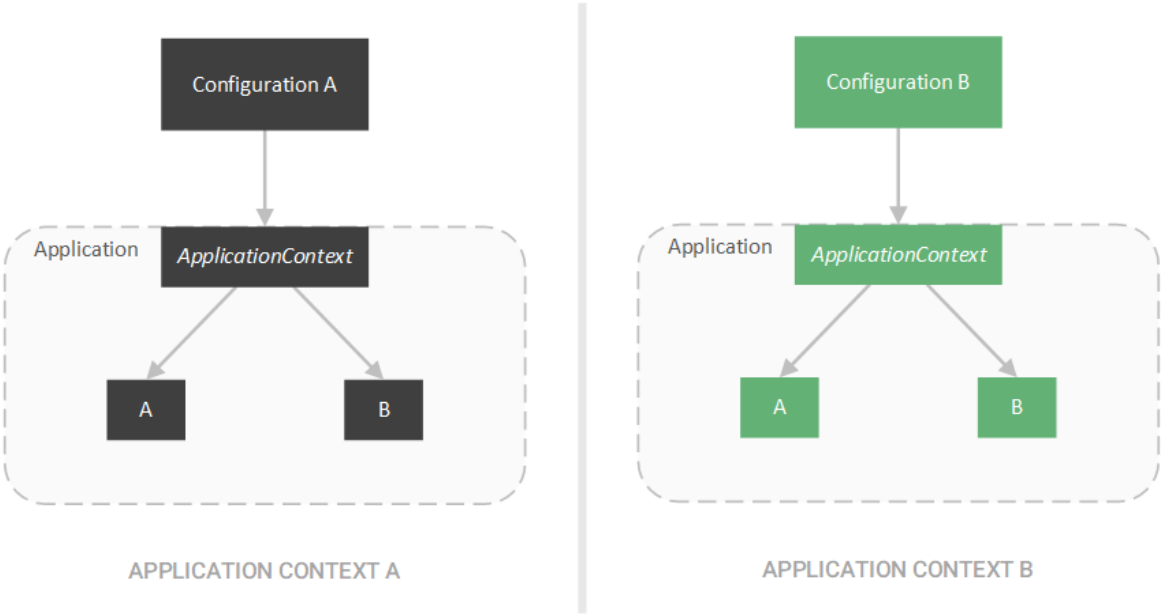
```
@Test
public void whenGetSimpleBean_thenReturnConstructedBean() {

   ApplicationContext context = new
AnnotationConfigApplicationContext(ApplicationConfig.class);

   Foo foo = context.getBean(Foo.class);

   assertNotNull(foo);
}

@Test
public void whenGetPrototypeBean_thenReturnConstructedBean() {

   String expectedName = "Some name";
   ApplicationContext context = new
AnnotationConfigApplicationContext(ApplicationConfig.class);

   Bar bar = context.getBean(Bar.class, expectedName);

   assertNotNull(bar);
   assertThat(bar.getName(), is(expectedName));
}
```

getBean factory metodunu kullanarak class tipini ve constructor parametrelerini geçip konfigüre ettiğimiz beanleri oluşturabiliriz.

2.2 External Configuration

Bu pattern çok yönlüdür çünkü uygulamanın davranışını harici konfigürasyona dayalı olarak tamamen değiştirebiliriz.

Uygulamadaki autowired nesnelerin uygulamasını değiştirmek istersek kullandığımız ApplicationContext' implementasyonunu genişletebiliriz.

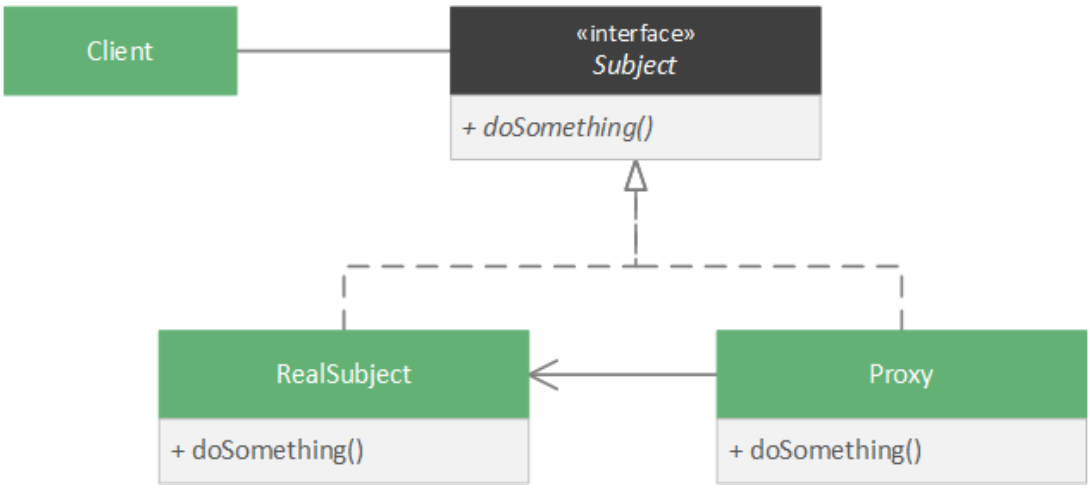


Örneğin, AnnotationConfigApplicationContext öğesini ClassPathXmlApplicationContext gibi XML tabanlı bir yapılandırma sınıfıyla değiştirebiliriz :

```
@Test
public void
givenXmlConfiguration_whenGetPrototypeBean_thenReturnConstructedBean() {

   String expectedName = "Some name";
   ApplicationContext context = new
ClassPathXmlApplicationContext("context.xml");

   // Same test as before ...
}
```

3. Proxy Pattern

Proxy'ler dijital dünyamızda kullanışlı bir araçtır ve bunları çok sık yazılım dışında (ağ proxy'leri gibi) kullanırız. Kodda, proxy modeli , bir nesnenin - proxy'nin - başka bir nesneye - özne veya hizmete - erişimini kontrol etmesine izin veren bir tekniktir.



3.1 Transactions

Bir proxy oluşturmak için, öznemizle aynı arayüzü uygulayan ve özneye bir referans içeren bir nesne oluşturuyoruz.

Daha sonra öznenin yerine proxy'yi kullanabiliriz.

Spring'de beanler, alttaki bean'e erişimi kontrol etmek için proxy'ye alınır. İşlemleri kullanırken bu yaklaşımı görüyoruz

```
@Service
public class BookManager {

   @Autowired
   private BookRepository repository;

   @Transactional
   public Book create(String author) {
      System.out.println(repository.getClass().getName());
      return repository.create(author);
   }
}
```

BookManager sınıfımızda, create yöntemine @Transactional anotasyonu ekliyoruz. Bu anotasyon, Spring'e oluşturma yöntemimizi atomik olarak yürütmesini söyler. Bir proxy olmadan Spring, BookRepository çekirdeğimize erişimi kontrol edemez ve

işlem tutarlılığını sağlayamazdı.

3.2 CGLib Proxies

Bunun yerine Sğring BookRepository bean'ini çevreleyen bir proxy oluşturur. Bizim bean'imizin create metodunun otomatikk olarak çalıştırılmasını sağlar.

BookManager#create yöntemimizi çağırdığımızda çıktıyı görebiliriz:

Tipik olarak, standart bir BookRepository nesne kimliği görmeyi bekleriz; bunun yerine bir EnhancerBySpringCGLIB nesne kimliği görüyoruz .

Sahne arkasında Spring, BookRepository nesnemizi EnhancerBySpringCGLIB nesnesi olarak içine sardı . Spring böylece BookRepository nesnemize erişimi kontrol eder (işlemsel tutarlılığı sağlar).

Spring genellikle iki tür proxy kullanır :

- CGLib Proxy'leri – Sınıfları proxy yaparken kullanılır
- JDK Dinamik Proxy'ler – Arayüzleri proxy yaparken kullanılır

Temel proxy'leri ortaya çıkarmak için işlemleri kullanırken, Spring, bir bean'e erişimi kontrol etmesi gereken herhangi bir senaryo için proxy'leri kullanacaktır .

4. Template Method Pattern

Birçok frameworkte, kodun önemli bir kısmı boilerplate koddur.

Örneğin, bir veritabanında bir sorgu yürütülürken, aynı adımlar dizisi tamamlanmalıdır:

1 - bağlantı kurun
2 - Sorguyu çalıştır
3 - Temizleme gerçekleştirin
4 - bağlantıyı kapat

Bu adımlar, şablon yöntemi deseni için ideal bir senaryodur .

4.1 Templates *& Callbacks*

Template method pattern'i bazı eylemler için gerekli adımları tanımlayan, standart adım adımlarını uygulayan ve özelleştirilebilir adımları soyut olarak bırakan bir tekniktir. Alt sınıflar daha sonra bu soyut sınıfı uygulayabilir ve eksik adımlar için somut bir uygulama sağlayabilir.

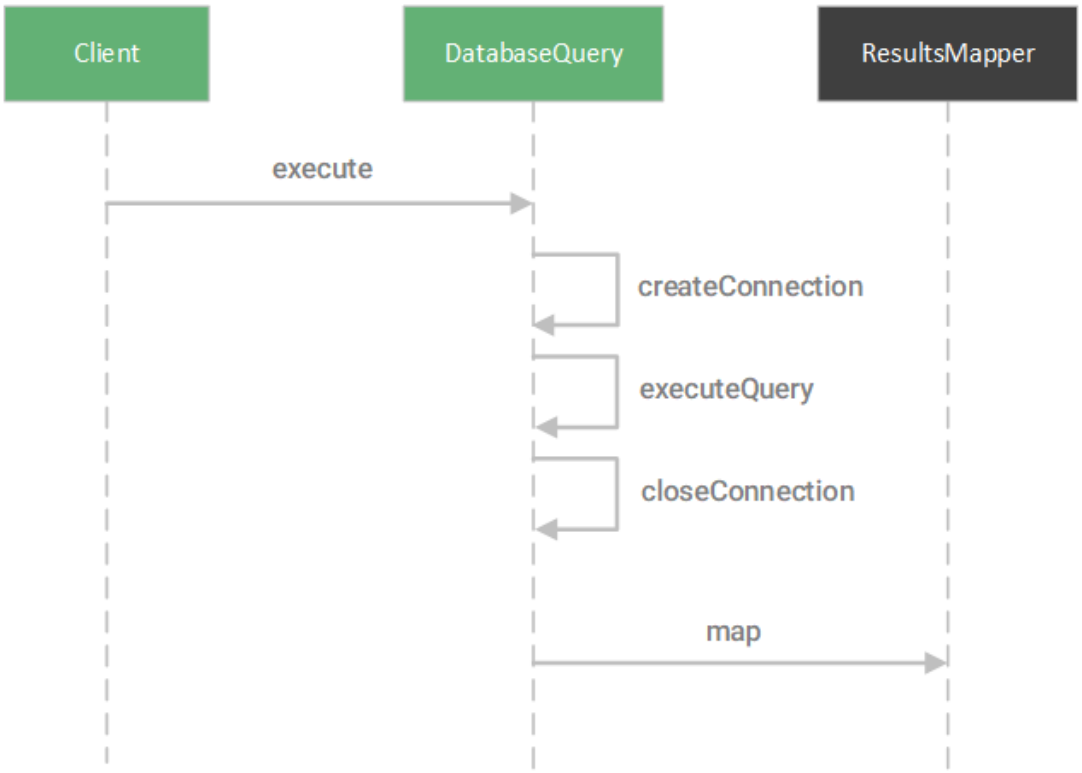
Veritabanı sorgumuz durumunda bir şablon oluşturabiliriz:

```
public abstract DatabaseQuery {

   public void execute() {
      Connection connection = createConnection();
      executeQuery(connection);
      closeConnection(connection);
   }

   protected Connection createConnection() {
      // Connect to database...
   }

   protected void closeConnection(Connection connection) {
      // Close connection...
   }

   protected abstract void executeQuery(Connection connection);
}
```

Alternatif olarak, bir callback metodu sağlayarak eksik adımı sağlayabiliriz.

Callback metodları, öznenin istemciye istenen bazı eylemlerin tamamlandığını bildirmesine izin veren bir yöntemdir .

Bazı durumlarda, özne, sonuçları eşleme gibi eylemleri gerçekleştirmek için bu geri aramayı kullanabilir.



Örneğin, bir executeQuery metoduna sahip olmak yerine, sonuçları işlemek için metoda bir sorgu stringi ve bir callback metodu sağlayabiliriz.

İlk olarak, bir Results nesnesini alan ve onu T türünde bir nesneyle eşleyen callback metodu oluşturuyoruz :

```
public interface ResultsMapper<T> {
   public T map(Results results);
}
```

Sonra bu callback'i kullanmak için DatabaseQuery sınıfımızı değiştiriyoruz:

```
public abstract DatabaseQuery {

   public <T> T execute(String query, ResultsMapper<T> mapper) {
      Connection connection = createConnection();
      Results results = executeQuery(connection, query);
      closeConnection(connection);
      return mapper.map(results);
   ]

   protected Results executeQuery(Connection connection, String query) {
      // Perform query...
   }
```

```
}
```

Bu callback mekanizması Spring'in JdbcTemplate classıyla kullandığı yaklaşımdır.

4.2 JdbcTemplate

JdbcTemplate sınıfı, bir sorgu dizesini ve ResultSetExtractor nesnesini kabul eden sorgu yöntemini sağlar

```
public class JdbcTemplate {

   public <T> T query(final String sql, final ResultSetExtractor<T> rse) throws
DataAccessException {
      // Execute query...
   }

   // Other methods...
}
```

ResultSetExtractor , sorgunun sonucunu temsil eden ResultSet nesnesini T türünde bir etki alanı nesnesine dönüştürür :

```
@FunctionalInterface
public interface ResultSetExtractor<T> {
   T extractData(ResultSet rs) throws SQLException, DataAccessException;
}
```

Spring, daha spesifik callback arayüzleri oluşturarak ortak kodu daha da azaltır. Örneğin, RowMapper arabirimi, tek bir SQL verisi satırını T türünde bir etki alanı nesnesine dönüştürmek için kullanılır .

```
@FunctionalInterface
public interface RowMapper<T> {
   T mapRow(ResultSet rs, int rowNum) throws SQLException;
}
```

Spring, RowMapper interface'ini beklenen ResultSetExtractor'a uyarlamak için RowMapperResultSetExtractor classını oluşturur:

```
public class JdbcTemplate {

   public <T> List<T> query(String sql, RowMapper<T> rowMapper) throws
DataAccessException {
      return result(query(sql, new RowMapperResultSetExtractor<>(rowMapper)));
   }

   // Other methods...
```

```
}
```

Tüm bir ResultSet nesnesini dönüştürmek için logic sağlamak yerine, satırlar üzerinde yineleme de dahil olmak üzere, tek bir satırın nasıl dönüştürüleceğine dair logic sağlayabiliriz:

```
public class BookRowMapper implements RowMapper<Book> {

  @Override
  public Book mapRow(ResultSet rs, int rowNum) throws SQLException {

    Book book = new Book();

    book.setId(rs.getLong("id"));
    book.setTitle(rs.getString("title"));
    book.setAuthor(rs.getString("author"));

    return book;
  }
}
```

Bu dönüştürücü ile daha sonra JdbcTemplate kullanarak bir veritabanını sorgulayabilir ve ortaya çıkan her satırı eşleyebiliriz:

```
JdbcTemplate template = // create template...
template.query("SELECT * FROM books", new BookRowMapper());
```

Spring, JDBC veritabanı yönetiminin yanı sıra aşağıdakiler için şablonlar kullanır:

- Java Message Service (JMS)
- Java Persistence API (JPA)
- Hibernate (now deprecated)
- Transactions