

# Spring Framework

**KurumsalJava.com**  
**KurumsalJavaAkademisi.com**

Özcan Acar  
Bilgisayar Mühendisi  
<http://www.ozcanacar.com>

# Giriş

Son yılların en popüler Java frameworklerinden birisi olan Spring<sup>1</sup> ile kurumsal yapıların ihtiyaç duydukları altyapıları hafif (light-weight) çözümlerle oluşturmak mümkündür. Spring ile oluşturulan programlarda EJB<sup>2</sup> teknolojisinden tanıdığımız deklaratif transaksyon yönetimi, aplikasyon güvenliği, ORM persistens gibi teknolojiler bir EJB container serverine ihtiyaç duyulmadan uygulanabilir. Spring ile oluşturulan programların test edilmeleri daha kolaydır ve Spring framework XP projelerinde test güdümlü yazılımı (TDD) desteklemektedir. Sunduğu Spring MVC web frameworkü ile web tabanlı programlar hazırlamak mümkündür.

Spring framework programcı için birçok işlemi basitleştirir ve kullanımını kolaylaştırır. Spring üç ayaktan oluşur:

- Spring basit ve sadeleştirilmiş bir API sunarak, birçok open source ürünün kullanımını ve entegrasyonunu kolaylaştırır. Son yıllarda Java platformunun (Java SE, Java EE) kullanımı, sunmuş olduğu detaylı API'lerden dolayı zorlaşmıştır. Spring kendi API'leri ile bu sorunu çözer.
- Spring içinde Dependency Injection (bağımlılıkların enjekte edilmesi) metodu kullanarak, nesneler arası bağlar XML konfigürasyon dosyaları üzerinden otomatik olarak gerçekleştirilir. Örneğin ClassA isimindeki bir sınıf ClassB tipinde bir değişkene sahip ise, bu bağımlılık ClassA sınıfından bir nesne oluşturulduktan sonra Spring tarafından göz önünde bulundurulur. Spring otomatik olarak ClassB sınıfından bir nesne oluşturarak, ClassA sınıfından oluşturduğu nesneye enjekte eder. İki sınıf arasındaki bağ böylece Spring tarafından Dependency Injection metodu ile oluşturulmuş olur.
- Spring AOP (aspect oriented programming <sup>3</sup>) tarzı program yazılımını destekler. Genelde transaksyon yönetimi, logging ve güvenlik gibi program parçaları kodun birçok yerinde kullanılır. AOP ile aslında program mantığının bir parçası olmak zorunda olmayan bu metod ve modüller merkezi bir yerde toplanarak, programdan bağımsız bir şekilde implemente edilir.

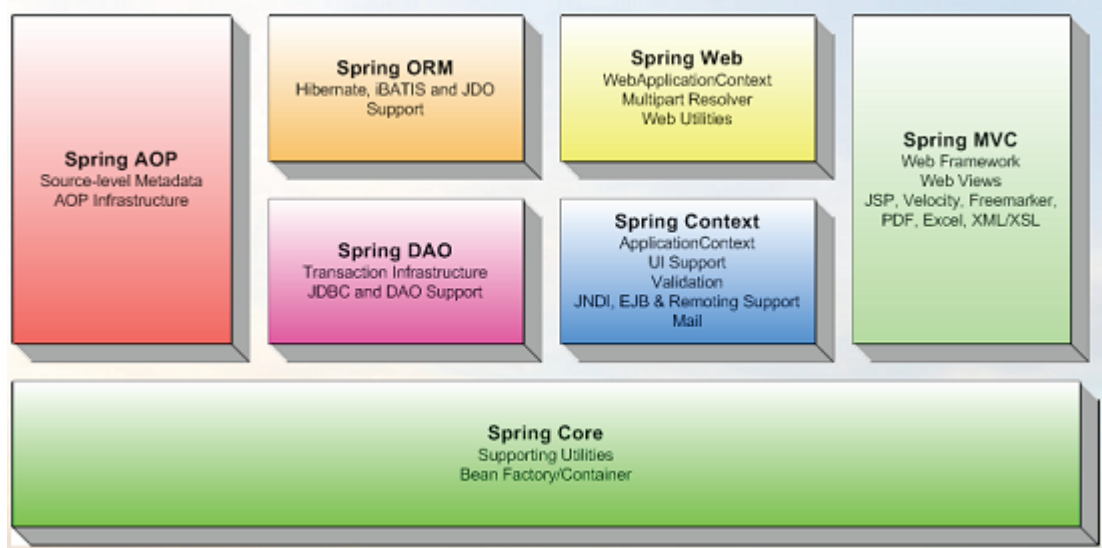
Spring'in amacı programcının hayatını kolaylaştırmak ve ona birçok kullanım özgürlüğü tanımaktır. Bu sebepten dolayı Spring frameworkünü oluşturan parçalar birbirinden bağımsız bir şekilde kullanılabilir.

---

<sup>1</sup> Bakınız: <http://www.springframework.org>

<sup>2</sup> Bakınız: <http://java.sun.com/products/ejb/>

<sup>3</sup> Bakınız: <http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html>



**Resim 12.1** Spring Framework

## Spring Modülleri

Spring, birbirinden bağımsız kullanılabilen modüllerden oluşmaktadır. Spring modül yapısı resim 10.1 de yer almaktadır.

### Spring Core:

Spring core paketi, Spring'in temelini oluşturur. Dependency injection gibi temel fonksiyonlar Spring Core içinde implemente edilmiştir.

### Spring AOP:

Aspect oriented programming bu modülde implemente edilmiştir. Spring'in AOP özellikleri direk kullanılsada, deklaratif transaksion ve güvenlik uygulamalarında Spring AOP Spring Core tarafından dolaylı olarak kullanılır.

### Spring MVC:

Spring MVC (Model-View-Controller) Spring baz alınarak web tabanlı programlar oluşturulabilir. Spring MVC bir web frameworküdür.

### Spring Context:

ApplicationContext ve WebApplicationContext gibi sınıflar bu modülde implemente edilmiştir. XML dosyalarından bulunan Spring bean tanımlamalarını okumak ve Spring bean nesnelerini oluşturmak için kullanılırlar.

### Spring DAO (Database Access Objec):

Bilgibankaları üzerinde işlem yapmak için kullanılan JDBC teknolojisini kullanımda daha basit hale getiren sınıflar bu modül içinde yer alır.

### **Spring ORM (Object Relational Mapping):**

Bu modül Hibernate, JDO, TopLink ve IBatis gibi popüler ORM frameworkler ile entegrasyonu sağlamaktır.

## **Spring Kullanımının Avantajları**

XP projelerinde temel altyapı ihtiyaçlarını karşılamak için Spring kullanılabilir. Bunlar otomatik transaksyon yönetimi, sistem güvenliği, Hibernate gibi bir ORM frameworkünün entegrasyonu, web servisi oluşturulması ve kullanımı, EJB ve RMI serverlerinin kullanımı gibi konular olabilir. Spring değişik komponentlerin entegrasyonu için gerekli altyapı hizmetlerini sunar. Spring kullanımı sonucu elde edilen avantajları şu şekilde sıralayabiliriz:

- Basit Java sınıfları (POJO – Plain Old Java Object) kullanılarak yazılım yapılabilir. Bu sınıfların test edilmesi, debugging yöntemiyle hataların aranması ve kurulumu (deployment) daha kolaydır. EJB teknolojisinde olduğu gibi bir EJB container serverine ihtiyaç duyulmamaktadır.
- Spring MVC ile web tabanlı programlar oluşturmak mümkündür. Web uygulamalarında deklaratif transaksyon yönetimi, güvenlik ve bilgibankası bağlantısı gibi temel altyapı ihtiyaçları Spring ve Spring MVC tarafından sağlanır.
- Spring tekeri yeniden icat etmeden, piyasadaki mevcut teknolojilerin entegre edilmesini kolaylaştırmaktadır.
- Spring modüler bir yapıya sahiptir. Proje gerekleri doğrultusunda belirli Spring modülleri kullanılabilir.
- Spring, konfigürasyon dosyalarında yapılacak ayarlamalar ile otomatik olarak Singleton nesneler oluşturabilir. Bunun için özel Singleton sınıfların oluşturulması gereği ortadan kalkar.
- Spring interface sınıflar için değişik türde implementasyon yapılmasını ve kullanım esnasında konfigürasyon dosyaları üzerinden implementasyon sınıf seçimi kolaylaştırır. Bu açıdan bakıldığında tasarım şablonlarının kullanımı Spring aracılığıyla daha kolay hale gelmektedir.
- Sunduğu Servlet Mock sınıfları ile, Spring MVC ile oluşturulan web tabanlı programların Tomcat gibi bir Servlet container serverinden bağımsız olarak test edilmesini kolaylaştırır.

## **Dependency Injection**

Java gibi nesneye yönelik programlama (OO – Object Oriented) yapılabilen bir dilde, gerçek hayatta var olan nesneleri Java sınıfları aracılığıyla modelleriz. Bu sınıflardan Java programlarında kullanılan nesneler üretilir. Nesneler görevlerini yerine getirirken başka nesneleri kullanabilirler. Böylece nesneler arası bağımlılıklar oluşur. Java programcılarının çok sıklıkla karşılaştıkları NullPointerException, kullanılmak istenen nesnenin null değerine sahip olmasından kaynaklanmaktadır, yani new operatörü ile gerekli nesne oluşturulmamıştır. NullPointerException hatasını önlemek için bağımlılık duyulan nesnelerin oluşturulması

gerekmektedir. Bunu çoğu zaman aşağıdaki göreceğimiz örnekteki gibi elden yapmamız gerekmektedir.

#### Kod 12.1 ClassA.java - Dependency Injection örneği

```
package spring;

/**
 * Basit bir Java Bean
 * (POJO - Plain Old Java Object)
 *
 * @author Oezcan Acar
 */
public class ClassA
{
    /**
     * ClassA ile ClassB arasında bu sınıf
     * değişkeni üzerinden bağımlılık
     * vardır.
     */
    private ClassB classB;

    public ClassB getClassB()
    {
        return classB;
    }

    public void setClassB(ClassB classB)
    {
        this.classB = classB;
    }

    public void print(String msg)
    {
        classB.print(msg);
    }
}
```

ClassA isimindeki sınıfın ClassB tipinde bir sınıf değişkeni mevcuttur. Bu sebepten dolayı ClassA ile ClassB arasında bir bağımlılık oluşmaktadır. ClassA, ClassB olmadan görevini yerine getiremeyecektir. Bu sebepten dolayı ClassA sınıfından bir nesne oluşturulduğunda, classB sınıf değişkeninin değeri null olmamalıdır, aksi takdirde print() metodu bünyesinde bir NullPointerException oluşur.

#### Kod 11.2 ClassB.java - Dependency Injection örneği

```
package spring;

/**
 * Basit bir Java Bean
 * (POJO - Plain Old Java Object)
 *
 * @author Oezcan Acar
 */
```

```

*/
public class ClassB
{
    public void print(String msg)
    {
        System.out.println(msg);
    }
}

```

ClassB basit bir POJO (plain old java object) sınıftır ve print() isminde bir metoda sahiptir. ClassA'nın ClassB ye olan bağımlılığında dolayı, bu sınıfların kullanımı aşağıdaki şekilde olmak zorundadır:

### Kod 11.3 Test.java - Dependency Injection örneği

```

package spring;

import org.javatasarim.spring.util.SpringUtil;

/**
 * Dependency Injection test sinifi
 *
 * @author Oezcan Acar
 */
public class Test
{

    public static void main(String[] args)
    {
        ClassA classA = new ClassA();
        ClassB classB = new ClassB();
        classA.setClassB(classB);
        classA.print("Hello Spring World");
    }
}

```

Örnekte görüldüğü gibi ClassA ve ClassB'den oluşan bir nesne ağı oluşturulmuştur. Birçok nesneden oluşan bir nesne ağına, bağımlılık duyulan nesnelerden sadece bir tanesinin değerinin null olması, NullPointerException oluşmasına sebep verecektir. Ayrıca nesne ağına yer alan nesnelerin new operatörü ile tek tek oluşturulması ve set() metotları ile ağı içine yerleştirilmeleri zaman alıcı ve hataya yol açabilecek bir yöntemdir. Spring dependency injection kullanılarak nesne ağının oluşturulması daha sadeleştirilebilir.

Spring bünyesinde, kullanılacak nesneler **applicationContext.xml** ismini taşıyan bir XML dosyasında tanımlanır. Bu dosya içinde her sınıf için bir <bean/> tagı yer alır. Bir sonraki tabloda applicationContext.xml dosyasının içeriğini görmekteyiz.

### Kod 11.4 applicationContext.xml - Dependency Injection örneği

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>

    <bean id="ClassA"
        class="org.javatasarim.spring.dependencyinjection.ClassA">
        <property name="classB" ref="ClassB" />
    </bean>

    <bean id="ClassB"
        class="org.javatasarim.spring.dependencyinjection.ClassB">
    </bean>

</beans>
```

ClassA ve ClassB isimlerinde iki Spring Bean tanımlıyoruz. Spring'in ClassA tipi bir nesne oluşturabilmesi için bu sınıfın paket ismini tanıması gerekmektedir. ClassA ile ClassB arasındaki bağı oluşturmak için <property/> tagı kullanılır. Bu tag bünyesinde "name" elementi ile enjekte edilmesi gereken değişken tanımlanır. <property name="classB" ref="ClassB"/> şeklinde bir tanımlama şu anlama gelmektedir: "ClassA sınıfında classB isminde bir değişken vardır. ClassA'dan bir nesne oluştururken, classB ismindeki değişkeninde oluşturulması gerekmektedir. classB nesnesi oluşturulurken referans olarak (ref="ClassB") ClassB ismini taşıyan Spring Bean (<bean id="ClassB".../> kullanılabilir."

Spring bu tanımlamalar yardımı ile ClassA isminde bir nesne oluştururken, bu sınıfın değişkeni olan classB'yide oluşturup, otomatik olarak ClassA nesnesine enjekte edecektir. Bu işleme *dependency injection yani bağımlılıkların enjekte edilmesi* adı verilir. Spring'in nasıl kullanılabileceğini yeni bir örnekte görelim:

#### Kod 11.5 TestSpring.java - Dependency Injection örneği

```
package spring;

import org.javatasarim.spring.util.SpringUtil;

/**
 * Spring Dependency Injection
 * test sinifi
 *
 * @author Oezcan Acar
 */
public class TestSpring
{
    public static void main(String[] args)
    {
        /**
         * SpringUtils sinifi araciligi ile bir Spring Bean
         * olan ClassA sinifindan classA nesnesi edinilir.
         */
        ClassA classA = (ClassA)SpringUtil.getBean("ClassA");

        /**
```

```

        * Eger classA nesnesi Spring üzerinden dependency
        * injection ile olusturulmamis olsa idi, asagida
        * yeralan classA.print() metodu NullPointerException
        * olustururdu, çünkü ClassA.print() metodu bünyesinde
        * ClassB tipinde bir degisken kullaniliyor. classB
        * null oldugundan, classA.print() NullPointerException
        * verir.
        */
classA.print("Hello Spring World");
    }
}

```

### Kod 11.6 SpringUtil.java - Dependency Injection örneği

```

package spring;

import
org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * SpringUtils araciligi ile Spring ApplicationContext'in
 * tanimlandigi applicationContext.xml dosyasi
 * edinilir. getBean(String beanName) ile bu dosyada
 * tanimlanmis bir Spring Bean nesnesine ulasilir.
 *
 * @author Oezcan Acar
 */
public class SpringUtil
{
    private static final ClassPathXmlApplicationContext factory =
        new
ClassPathXmlApplicationContext("spring/applicationContext.xml");

    public static Object getBean(String name)
    {
        return factory.getBean(name);
    }
}

```

TestSpring sınıfı işlev itibarıyla Test sınıfı ile aynı yapıdadır. Aradaki tek fark, TestSpring bünyesinde Spring frameworkünün ve beraberinde getirdiği dependency injection metodunun kullanılıyor olmasıdır. Görüldüğü gibi SpringUtil.getBean("ClassA") ile applicationContext.xml bünyesinde tanımlanmış olan ClassA Spring Bean kullanılmaktadır. Spring burada genel olarak Factory tasarım şablonunda da olduğu gibi nesne üreten bir fabrika olarak düşünülebilir. Spring bu durumda otomatik olarak ClassA ve ClassB den oluşan nesne ağını oluşturur ve kullanıma sunar. Son satırda yeralan classA.print() metodu, classB nesnesinin otomatik olarak enjekte edilmesinden dolayı NullPointerException hatası vermeyecektir.

## Spring ile Tasarım Sablonu Kullanımı



Birçok tasarım şablonunda interface sınıflarının önemli bir yer almaktadır. Interface sınıflarının kullanılmasının amacı, kullanıcı sınıf ile, interface implementasyonu arasında esnek bir bağ oluşmasını sağlamaktır. Kullanıcı sınıf, sadece interface bünyesinde bulunan metot isimlerine tanımak zorundadır. Bu metotların nasıl ve hangi sınıf tarafından implemente edildiği kullanıcı sınıf için önem taşımamaktadır. Kullanıcı sınıfın hangi implementasyon sınıfını kullandığını bilmek zorunda olmayışı, bu iki sınıf arasında esnek bir bağın oluşmasını sağlar.

Spring bu açıdan bakıldığında interface sınıfların kullanımı için biçilmiş bir kaftandır. Spring XML dosyasında (applicationContext.xml) kullanılması gereken implementasyon sınıfını tanımlayarak, kullanıcı sınıfı etkilemeden sistem konfigürasyonunu değiştirebiliriz.

Aşağıda yer alan örnekte bir posta kutusunda (Mailbox) yaralan mektuplara (Mail) erişmek için Dao tasarım şablonu kullanılmaktadır. Posta kutusu ve barındırdığı mektuplar bir bilgibankasında yer almaktadır. Bilgibankasına erişim Dao tasarım şablonu ile daha transparan hale getirilmiş olur.

#### Kod 11.7 Dao interface sınıfı

```
package spring;

import java.util.ArrayList;

/**
 * Dao interface sinifi
 *
 * @author Oezcan Acar
 */
public interface Dao
{
    ArrayList<Mail> getMails();
}
```

İlk olarak Dao isminde bir interface sınıf tanımlıyoruz. Bu sınıf bünyesinde, getMails() metodu yer almaktadır. Kullanıcı sınıflar sadece Dao interface sınıfını tanımak zorunda bırakıldıkları sürece, hangi implementasyonun kullanıldığı onlar için önem taşımayacaktır.

#### Kod 11.8 JdbcDaoImpl.java - Dao implementasyonu

```
package spring;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;

/**
 * Jdbc DAO implementasyonu
 *
 * @author Oezcan Acar
 */
```

```

public class JdbcDaoImpl implements Dao
{
    public ArrayList<Mail> getMails()
    {
        ArrayList<Mail> mails = new ArrayList<Mail>();
        Connection con = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try
        {
            con = getConnection();
            pstmt = con.prepareStatement("select * from mail");
            rs = pstmt.executeQuery();

            while(rs.next())
            {
                Mail mail = new Mail();
                mail.setSender(rs.getString(1));
                mail.setSubject(rs.getString(2));
                mail.setBody(rs.getString(3));
                mails.add(mail);
            }
        }
        catch (Exception e)
        {
            throw new RuntimeException(e);
        }
        finally
        {
            try
            {
                if(con != null && !con.isClosed())
                {
                    con.close();
                }
            }
            catch (SQLException e)
            {
                e.printStackTrace();
            }
        }
        return mails;
    }

    private Connection getConnection()
    {
        // kullanılan bilgibankasına
        // göre implemente edilmesi gerekiyor
        return null;
    }
}

```

Gerçek bir bilgibankası (Oracle, Mysql gibi) kullanabilmek için JdbcDaoImpl isminde, Dao interface sınıfını implemente eden bir sınıf oluşturduk. Yukarıda yer alan bu sınıf Jdbc üzerinden mail tablosunda (select \* from mail) bulunan mektullara ulaşarak Mail nesneleri

oluşturur. Oluşturulan Mail nesneleri bir ArrayList içine yerleştirilerek, metodu kullanan sınıfa geri verilir.

Bunun yanısıra gerçek bir bilgibankası kullanma zorunluluğunu ortadan kaldırmak için, bir dummy (sahte) implementasyon oluşturulabilir. Aşağıda yeralan DummyDaoImpl gerçek bir bilgibankası kullanmak zorunda kalmadan, sistemin test edilmesi için kullanılabilir.

#### Kod 11.9 DummyDaoImpl.java - Dao implementasyonu

```
package spring;

import java.util.ArrayList;

/**
 * Dummy DAO implementasyonu
 *
 * @author Oezcan Acar
 */
public class DummyDaoImpl implements Dao
{
    public ArrayList<Mail> getMails()
    {
        ArrayList<Mail> mails = new ArrayList<Mail>();

        Mail mail1 = new Mail();
        mail1.setSender("Sender1");
        mail1.setSubject("subject1");

        mails.add(mail1);

        Mail mail2 = new Mail();
        mail2.setSender("Sender2");
        mail2.setSubject("subject2");

        mails.add(mail2);

        return mails;
    }
}
```

JdbcDaoImpl sınıfında olduğu gibi, DummyDaoImpl sınıfıda getMails() metodunun implementasyonunu yapmaktadır.

Buraya kadar bir interface sınıf ve iki değişik implementasyon sınıfını oluşturduk. Spring ile istenilen tipte bir implementasyonun kullanılabilmesi için aşağıdaki şekilde applicationContext.xml bünyesinde Dao Spring Bean'in oluşturulması gerekmektedir.

#### Kod 11.10 applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>

    <bean id="Dao"
        class="org.javatasarim.spring.dao.JdbcDaoImpl">
    </bean>

</beans>
```

### Kod 11.11 Test.java

```
package spring;

import java.util.ArrayList;
import org.javatasarim.spring.util.SpringUtil;

/**
 * Test programi.
 *
 * @author Oezcan Acar
 */
public class Test
{
    public static void main(String[] args)
    {
        Dao dao = (Dao)SpringUtil.getBean("Dao");
        ArrayList<Mail> mails = dao.getMails();
        for(int i=0; i < mails.size(); i++)
        {
            Mail mail = mails.get(i);
            System.out.println("Sender: " +mail.getSender());
            System.out.println("Subject: " +mail.getSubject());
        }
    }
}
```

Test.main() bünyesinde SpringUtils sınıfı yardımıyla Spring tarafından yönetilen Dao isimli bir Spring Bean ediniyoruz. Bu nesne, applicationContext.xml içinde tanımlamış olduğumuz Dao implementasyon sınıfından üretilmiş bir nesne olacaktır. JdbcDaoImpl implementasyon sınıfını kullandığımız için SpringUtil.getBean("Dao") bize bu sınıftan bir nesne üreterek geri verecektir.

Tasarım açısından Spring'in beraberinde getirdiği esnekliği Test sınıfında görmekteyiz. Test sınıfı ile Dao interface sınıfı arasında bir bağımlılık mevcuttur, çünkü Test sınıfı Dao interface sınıfını kullanmaktadır. Spring istenilen tipte bir implementasyon (JdbcDaoImpl) oluşturarak, Test sınıfına iletir. Test sınıfı sadece Dao interface sınıfını tanıdığı için, hangi implementasyonu kullandığını bilmek zorunda değildir. Bu sayede Test sınıfını etkilemeden kullanılan Dao implementasyonu her zaman değiştirilebilir. Spring, applicationContext.xml içinde tanımlanmış olan implementasyon sınıfını kullanarak, gerekli implementasyon nesnesini oluşturacak ve kullanıma sunacaktır.

Spring'in sağlamış olduđu diğ er entegratif özellikleri diğ er bir makalemizde yakından inceleyeceğiz.

*EOF ( End Of Fun)*

*Özcan Acar  
Bilgisayar Mühendisi  
KurumsalJava.com  
KurumsalJavaAkademisi.com*