# WEEK-3 YAZILI SORULAR

**Creational Design Pattern'lar incelenmelidir. Örneklerle anlatınız.**

Tasarım deseni, yazılım geliştiricilerin yazılım geliştirirken karşılaştıkları sorunlara karşı genel çözümlerdir. Yazılım geliştirdiğimiz sırada ortaya çıkan bir tasarım sorununu çözmek için kullanabileceğiniz önceden hazırlanmış planlardır. Tasarım desenleri 3 ana başlığa ayrılmaktadır: Creational patterns (Yaratımsal desenler) Structural patterns (Yapısal desenler) ve Behavioral patterns (Davranışsal desenler).
Yaratımsal desen nesneleri doğrudan new operatörü kullanarak oluşturmak yerine nesne oluşturma mantığını gizleyerek sınıflardan nesne oluşturmaya alternatif çözümler sunar. Yazdığımız uygulama akışında hangi nesneye ihtiyaç varsa onu oluşturmada esneklik ve kolaylık sağlar. Bunun için Singleton, Factory, Abstract Factory, Builder ve Prototype gibi yaratımsal tasarım desenleri geliştirilmiştir.

**Singleton Pattern:**

Singleton, bir nesnenin sadece bir örneğinin olduğundan emin olmak ve bu nesneye ihtiyacınız olduğunda kodunuzda her yerde aynı (ve tek örneğin) çağırılmasını sağlamak için kullanılır. Singleton sınıfı oluşturmak için sınıfın constructor (yapıcı metodu) private olarak tanımlanmalıdır. Bu sayede dışarıdan erişime kapatılır. Sınıfın türünde private (özel) bir nesne tanımlanır. Oluşturduğumuz nesneyi oluşturmamıza ve nesneye erişmemize izin veren bir statik fonksiyon oluşturulup. Fonksiyonun içerisine birden fazla nesne oluşturulmasını kısıtlayan bir koşul koyulur.

Singletons, veritabanlarıyla çalışırken kullanılabilir. Tüm istemciler için aynı bağlantıyı yeniden kullanırken veritabanına erişmek için bir bağlantı havuzu oluşturmak için kullanılabilirler. Örneğin,

```
class Database {
   private static Database dbObject;

   private Database() {
   }

   public static Database getInstance() {

      // create object if it's not already created
      if(dbObject == null) {
         dbObject = new Database();
      }

       // returns the singleton object
       return dbObject;
   }

   public void getConnection() {
       System.out.println("You are now connected to the database.");
   }
}

class Main {
   public static void main(String[] args) {
      Database db1;

      // refers to the only object of Database
      db1= Database.getInstance();
```

Programı çalıştırdığımızda çıktı şöyle olacaktır:

```
You are now connected to the database.
```

**Prototype Pattern:**

Bu yaklaşım elimizde var olan bir nesneden birden fazla kez kopyasını oluşturmamız gerektiğinde bizlere bu nesneyi tekrar tekrar sıfırdan oluşturmayıp, mevcut nesnenin klonlarını almamızı ele alır. Bu klonlama(kopyalama) işleminde, deep-copy yöntemi kullanılıyor. Yani bir nesne, birebir kopyalanarak yeni bir referans değişkene atılıyor. Prototip tasarım deseni örneğini gösteren örnek bir program. Employee sınıfı içerisinde çalışanların listesini tutuyor. Cloneable arayüzünü implement etmesiyle beraber istediğimiz klonlama işlemini gerçekleştireceğiz.

```
import java.util.ArrayList;
import java.util.List;

public class Employees implements Cloneable{

	private List<String> empList;

	public Employees(){
		empList = new ArrayList<String>();
	}

	public Employees(List<String> list){
		this.empList=list;
	}
	public void loadData(){
		//read all employees from database and put into the list
		empList.add("Pankaj");
		empList.add("Raj");
		empList.add("David");
		empList.add("Lisa");
	}

	public List<String> getEmpList() {
		return empList;
	}

	@Override
	public Object clone() throws CloneNotSupportedException{
			List<String> temp = new ArrayList<String>();
			for(String s : this.getEmpList()){
				temp.add(s);
			}
			return new Employees(temp);
	}

}
```

Override ettiğimiz clone metodu sayesinde mevcut çalışanların bulunduğu liste dışardan gelen listeye çalışanların eklenmesini sağlayacak ayrıca bize klon objesini geri döndürecek. Main sınıfımızda clone metodu sayesinde oluşan klon Employee objeleri oluşacak. Bu sayede istediğimiz işlemleri yapacağız.

```
import java.util.List;

import com.journaldev.design.prototype.Employees;

public class PrototypePatternTest {

	public static void main(String[] args) throws CloneNotSupportedException {
		Employees emps = new Employees();
		emps.loadData();

		//Use the clone method to get the Employee object
		Employees empsNew = (Employees) emps.clone();
		Employees empsNew1 = (Employees) emps.clone();
		List<String> list = empsNew.getEmpList();
		list.add("John");
		List<String> list1 = empsNew1.getEmpList();
		list1.remove("Pankaj");

		System.out.println("emps List: "+emps.getEmpList());
		System.out.println("empsNew List: "+list);
		System.out.println("empsNew1 List: "+list1);
	}

}
```

Programı çalıştırdığımızda çıktı şöyle olacaktır:

```
emps List: [Pankaj, Raj, David, Lisa]
empsNew List: [Pankaj, Raj, David, Lisa, John]
empsNew1 List: [Raj, David, Lisa]
```

**Builder Pattern:**

Sınıflardan nesneler yaratmak için constructorları kullanırız. Sınıfımızda bulunan field sayısı fazla olduğu durumlarda birden fazla constructora ihtiyaç duyabiliriz. Nesne üreteceğimiz zaman birden fazla parametre olacağı için karmaşıklık yaratabiliyor. Bu karmaşık yapıya çözüm bulmak için builder pattern güzel bir çözüm yolu sunuyor.

```java
public class Person {

    private String name, surname, address;

    public Person(String name, String surname, String address) {
        this.name = name;
        this.surname = surname;
        this.address = address;
    }
    // Getter & setter metodları
}
```

```java
public class Person {

    private String name, surname, address;

    public Person(Builder builder) {
        this.name = builder.name;
        this.surname = builder.surname;
        this.address = builder.address;
    }

    public String getName() {
        return name;
    }

    public String getSurname() {
        return surname;
    }

    public String getAddress() {
        return address;
    }

    public static class Builder{

        private String name, surname, address;

        public Builder(){ }

        public Builder name(String name){
            this.name = name;
            return this;
        }

        public Builder surname(String surname){
            this.surname = surname;
            return this;
        }

        public Builder address(String address){
            this.address = address;
            return this;
        }

        public Company build(){
            return new Company(this);
        }

    }

}
```

İnsan sınıfımızda ad, soyad ve adres gibi fieldlar var. Bu sınıftan nesne oluşturmak için 3 alanı da constructor içine göndermemiz gerekiyor. Fakat biz adresin olmadığı bir nesne üretmek istesek yeni bir constructor yaratmamız gerekecekti. Daha fazla fieldların olduğu bir sınıfta birden fazla constructor oluşturmamız gerekecekti bu da karmaşıklığa yol açacaktı. Builder pattern tam bu noktada devreye girerek bizi zahmetten kurtarıyor. Person sınıfımızın içinde static bir inner class var ve bunun üstünden asıl nesnemizi oluşturuyoruz. Nesne oluştururken yalnızca istediğimiz alanlar ile birlikte oluşturabiliyoruz.

```
Person person = new
Person.Builder().name("Tuğrul").surname("Bayrak").address("Türkiye").
build();
```

Eper builder patternı kullanmasak bu sınıftan istediğimiz nesneleri default constructor ile oluşturup daha sonra setter ve getter metodlarını kullanacaktık. Bu da tekrarlı işleme yol açtığı için kodun okunabilirliğini düşürüp zaman kaybına yol açacaktı.

**Factory Pattern:**

Factory Method tasarım kalıbı, kalıtımsal ilişkileri olan nesnelerin üretilmesi amacıyla kullanılır. Burada asıl amaç nesnelerin üretilmesi görevinin bir metoda verilmesidir.

Shape isminde bir interface oluşturalım, daha sonra bu arayüzden kalıtım yapan somut sınıfları yaratalım. Oluşturduğumuz sınıflarda draw() fonksiyonunu override edip metodun hangi sınıftan geldiğini belirtelim.

```
public interface Shape {
   void draw();
}
```

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

ShapeFactory adında bir sınıf oluşturup, istediğimiz bir Shape nesnesini oluşturmak için bu sınıfı kullanacağız. İhtiyaç duyduğumuz nesne türünü elde etmek için koşul ifadeleri kullanıyoruz.

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

Main metodunun bulunduğu sınıfta istediğimiz türden nesneleri üretmek için gerekli ifadeleri yazarken Shape arayüzünü kullanıyoruz. Ürettiğimiz nesnelerin draw metodunu çağırdığımızda arayüzden sınıfta override ettiğimiz ilgili sonucu verir.

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of square
      shape3.draw();
   }
}
```

Çıktı:

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

**Abstract Factory**

Factory patternda ilişkisel olan birden fazla nesne üretimini ortak bir arayüz sayesinde oluşturduğumuz bir sınıf üzerinden istediğimiz nesneleri elde ediyorduk. Abstract factory tam burada devreye girmektedir. İlişkisel olan birden fazla nesnenin üretimini, her nesnenin özel arayüzlerini tasarlayarak ilgili nesneleri üretmemizi sağlar. Bir Shape arayüzünü ve onu uygulayan somut Rectangle sınıfları oluşturuluyor ve draw metodu override ediliyor. Daha sonra soyut AbstractFactory sınıfını oluşturuyoruz. AbstractFactory sınıfını genişleten ShapeFactory sınıfı tanımlanıyor. Bir Factory oluşturucu/oluşturucu sınıfı FactoryProducer oluşturuluyor. Buradan Square, Rectangle nesneleri üretiliyor.

```java
public interface Shape {
   void draw();
}
```

```java
public class RoundedRectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside RoundedRectangle::draw() method.");
   }
}
```

*RoundedSquare.java*

```java
public class RoundedSquare implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside RoundedSquare::draw() method.");
   }
}
```

*Rectangle.java*

```java
public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

```java
public abstract class AbstractFactory {
   abstract Shape getShape(String shapeType) ;
}
```

Verilen bilgilere dayanarak somut sınıfın nesnesini oluşturmak için AbstractFactory'yi genişleten factory sınıfları oluşturuluyor.

```java
public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

```
public class RoundedShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new RoundedRectangle();
      }else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new RoundedSquare();
      }
      return null;
   }
}
```

Shape gibi bir bilgiyi ileterek spesifik factory almak için bir Factory sınıfı oluşturuluyor.

```
public class FactoryProducer {
   public static AbstractFactory getFactory(boolean rounded){
      if(rounded){
         return new RoundedShapeFactory();
      }else{
         return new ShapeFactory();
      }
   }
}
```

İstenen nesne ismi bilgisini ileterek somut sınıflarının nesnelerini elde etmek için AbstractFactory'yi almak için oluşturulan factory yapısı kullanılır.

```
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {
      //get shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory(false);
      //get an object of Shape Rectangle
      Shape shape1 = shapeFactory.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape1.draw();
      //get an object of Shape Square
      Shape shape2 = shapeFactory.getShape("SQUARE");
      //call draw method of Shape Square
      shape2.draw();
      //get shape factory
      AbstractFactory shapeFactory1 = FactoryProducer.getFactory(true);
      //get an object of Shape Rectangle
      Shape shape3 = shapeFactory1.getShape("RECTANGLE");
      //call draw method of Shape Rectangle
      shape3.draw();
      //get an object of Shape Square
      Shape shape4 = shapeFactory1.getShape("SQUARE");
      //call draw method of Shape Square
      shape4.draw();

   }
}
```

Program çıktısı:

```
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside RoundedRectangle::draw() method.
Inside RoundedSquare::draw() method.
```

**Java dünyasındaki framework'ler ve çözdükleri problemler nedir?**

Framework'ler bizlere uygulamalar geliştirirken zamandan tasarruf etmek ve kısayollar sağlamak için tasarlanmıştır. Framework'ler geliştirme sürecini kısaltır, kodu basitleştirmeye ve değiştirmeye izin verir.

Spring, Java ile kurumsal tabanlı uygulama geliştirmede kullanılan, geliştiricilerin işlerini hızlandıran ve kolaylaştıran bir açık kaynaklı bir Java platformudur. Spring, modüler yapıda olması ve kolay kullanılabilirliği sayesinde öne çıkmaktadır. Otomatik yapılandırma gibi özelliklerle Spring Boot, sizi kodlama ve gereksiz yapılandırma zahmetinden kurtarır.

Yapısında Inversion of Control (IoC) de bulunan Spring nesnelerin oluşturulması, kodun yaşam döngüsü, nesneler arası bağımlılıklar gibi birçok olayın yönetimini yazılım geliştiriciden alıp, Spring'e verir. Spring bütün işi halleder. Spring'in sahip olduğu IoC'nin en önemli parçalarından birisi, yazdığınız sınıfları ve bunların bağımlılıklarını sizin için yöneten bir Dependency Injection (bağımlılık enjeksiyonu) sahip olmasıdır. Bağımlılık enjeksiyonu ile nesneler arasındaki bağlar XML yapılandırma dosyaları ile otomatik olarak gerçekleşir. Bu da aslında bağımlılıkları ortadan kaldırır. Spring Boot'un birçok özelliği mikro servisleri oluşturmanızı ve yönetmenizi kolaylaştırır. Mikro servisleri oluşturmak kodunuzun bakımını, testini ve uyumluluğunu artırmak gibi birçok faydayı beraberinde getirir.

HelloWorld adında bir sınıf oluşturup message adında bir field tanımlayalım, daha sonra bu field'a ait getter setter metotlarını oluşturalım.

```
package com.tutorialspoint;

public class HelloWorld {
   private String message;

   public void setMessage(String message){
      this.message  = message;
   }
   public void getMessage(){
      System.out.println("Your Message : " + message);
   }
}
```

İlk adım, framework API ClassPathXmlApplicationContext() kullandığımız bir uygulama bağlamı oluşturmaktır. Bu API, bean yapılandırma dosyasını yükler ve sonunda sağlanan API'ye dayalı olarak, tüm nesneleri, yani yapılandırma dosyasında belirtilen bean'leri oluşturmayı ve başlatmayı sağlar.

İkinci adım, oluşturulan içeriğin getBean() yöntemini kullanarak gerekli çekirdeği elde etmek için kullanılır. Bu yöntem, en sonunda gerçek nesneye dönüştürülebilen genel bir nesneyi döndürmek için bean kimliğini kullanır. Bir nesneye sahip olduğunuzda, herhangi bir sınıf yöntemini çağırmak için bu nesneyi kullanabilirsiniz.

```
package com.tutorialspoint;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainApp {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("Beans.
      HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
      obj.getMessage();
   }
}
```

Beans.xml, farklı bean'lere benzersiz kimlikler atamak ve Spring kaynak dosyalarından herhangi birini etkilemeden farklı değerlere sahip nesnelerin oluşturulmasını kontrol etmek için kullanılır. Örneğin, aşağıdaki dosyayı kullanarak "message" değişkeni için herhangi bir değer iletebilir ve HelloWorld.java ve MainApp.java dosyalarını etkilemeden farklı mesaj değerleri yazdırabilirsiniz.

```
<?xml version = "1.0" encoding = "UTF-8"?>

<beans xmlns = "http://www.springframework.org/schema/beans"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

   <bean id = "helloWorld" class = "com.tutorialspoint.HelloWorld">
      <property name = "message" value = "Hello World!"/>
   </bean>

</beans>
```

Spring uygulaması belleğe yüklendiğinde, Framework, tanımlanan tüm çekirdekleri oluşturmak için yukarıdaki yapılandırma dosyasını kullanır ve bunlara <bean> etiketinde tanımlandığı gibi benzersiz bir kimlik atar. Programımızı çalıştırdığımızda aşağıdaki mesajı alacağız ve uygulamamızın çalıştığını göreceğiz.

```
Your Message : Hello World!
```

Hibernate Java geliştiriciler için geliştirilmiş bir ORM kütüphanesidir. Nesne yönelimli modellere göre veri tabanı ile olan ilişkiyi sağlayarak, veri tabanı üzerinde yapılan işlemleri kolaylaştırmakla birlikte kurulan yapıyı da sağlamlaştırmaktadır.

Hibernate yalnızca Java sınıflarından veri tabanı tablolarına veya Java veri tiplerinde SQL veri tiplerine dönüşümü yapmaz. Hibernate veri sorgulama ve veri çekme işlemlerini de kullanıcı için sağlar. Bu özellikleriyle Hibernate geliştirme kolaylığı ve zamandan kazanç sağlar.

Verileri işlemek için kullandığımız temel CRUD (create, read, update, delete) işlemleri JDBC ile yapıldığında satırlarca kod yazılıyor.

```
1  statement = conn.createStatement();
2             String query = "Select * From worker";
3
4             ResultSet rs = statement.executeQuery(query);
5
6             while (rs.next()) {
7                 int id = rs.getInt("id");
8                 String name = rs.getString("name");
9                 String surname = rs.getString("surname");
10                String department = rs.getString("department");
11                int salary = rs.getInt("salary");
12        }
```

Örnekte gördüğünüz select işleminde, birçok satır kod yazıyoruz. Veriler ResultSet şeklinde geldiği için ayrıca onun kontrolünü de yapıyoruz. Dolayısıyla projemize satırlarca kod yüklenmeye devam ediyor. Bu da projemize yük olmakla birlikte zamanımızı da ciddi oranda etkiliyor.

```
List<City> cities = session.createQuery("from City").getResultList();

            for(City city:cities) {
                System.out.println(city.getName());
            }
```

Farkettiyseniz Hibernate sorgusu, bir SQL sorgusuna çok benziyor. Veri tabanındaki tablolarımızla, tablolarımızdaki bilgileri tutan sınıfları eşleştirip; sınıflar üzerinden veri tabanındaki nesneleri hızlıca map ederek yani ilişkilendirerek verilere hızlı bir şekilde CRUD işlemleri uygulayabilmemizi sağlıyor.

```
@Entity
@Table(name="city")
public class City {

    @Id
    @Column(name="ID")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;

    @Column(name="name")
    private String name;

    @Column(name="countrycode")
    private String countryCode;

    @Column(name="district")
    private String district;

    @Column(name="population")
    private int population;
```

JDBC ile insert işlemi;

```
statement.executeUpdate("INSERT INTO admin VALUES ('Emrecan', 'Ay')");
```

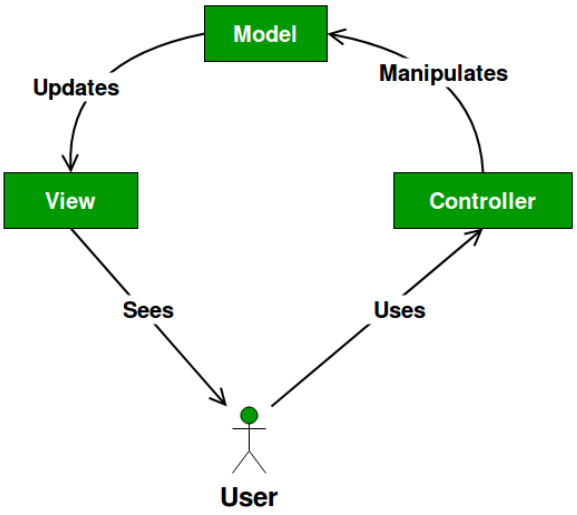
Hibernate ile insert işlemi;

```
session.save(admin);
```

**Spring frameworkünün kullandığı design patternlar neler?**

Spring, kurumsal Java (J2EE) için bir uygulama geliştirme çerçevesidir. Minimum kodlama ile web uygulamaları geliştirmek için esnek bir platform sağlar. Spring framework, aşağıdaki tasarım desenlerini kullanmaktadır.

Model-View-Controller (MVC) yazılım tasarım modeli, bir yazılım uygulamasındaki endişeleri ayırmak için bir yöntemdir. Prensip olarak, uygulama mantığı veya denetleyici, bilgileri kullanıcıya göstermek için kullanılan teknolojiden veya görünüm katmanından ayrılır. Model, denetleyici ve görünüm katmanları arasında bir iletişim aracıdır.




Model – Verilerin modellendiği genellikle sıradan Java sınıflarının (POJO) kullanıldığı bölümdür.

View – Görünüm, modelin verilerini kullanıcıya sunar. Görünüm, modelin verilerine nasıl erişileceğini bilir, ancak bu verilerin ne anlama geldiğini veya kullanıcının onu manipüle etmek için ne yapabileceğini bilmez.

Controller – Gelen isteğe göre genellikle model içerisinde yer alan verileri kullanarak işlem yapan ve sonucu view katmanına ileten bölümdür.

Spring Web MVC veya daha bilinen adıyla Spring MVC geliştirme sırasında MVC tasarım mimarisini kullanarak kolay, anlaşılır uygulama geliştirmeyi sağlar.

Spring framework, nesnenin yaratılmasından, nesnelerin birbirine bağlanmasından, bu nesnelerin yapılandırılmasından ve bu nesnelerin yaratılışlarından tamamen yok olana kadar tüm yaşam döngüsünün ele alınmasından sorumlu bir IOC kapsayıcısına sahiptir. Inversion of control bir yazılım tasarım prensibidir. Ioc ile Uygulama içerisindeki obje instance'larının yönetimi sağlanarak, bağımlılıklarını en aza indirgemek amaçlanmaktadır.

Uygulamamızdaki bağımlılıkların oluşturulmasını ve yönetilmesinin kullanıcıdan framework'e devredilmesi olarak da açıklanabilir. Üzerinde geliştirme yaptığımız framework'ler kullanıcıların yazdığı kodu çalıştırmak için gerekli kaynakları ve çalışması gereken metotları oluşturup yönetir. Uygulamayı çalıştırdığımızda framework yazılan kodları çağırır ve çalıştırır, daha sonra kontrol tekrar framework'e geçer bu olayın tamamına Inversion of Control diyoruz. Spring container, bir uygulamada mevcut olan bileşenleri yönetmek için kullanılan Dependency Injection (Bağımlılık Enjeksiyonuna) sahiptir. Bu tür nesneler Spring Bean olarak bilinir.

Dependency Injection aslında IoC'nin implement edilmesini sağlayan bir pattern'dır ve programımızın ihtiyacı olan bağımlılıkları sağlaması amaçlanır. Kod içerisine eklediğimiz bağımlılıkları bizim için getirir ve getirdiği bilgileri kodumuza ekler. Spring Framework kullanarak Dependency Injection sağlamanın 3 farklı yolu var.

Örneğin bir servis sınıfımız olsun ve bu servis sınıfımızın bağımlı olduğu bazı repository'leri ya da farklı servisleri olsun. Constructor ile Dependency Injection yapmayı planlıyorsak bunu değişkenlerimizin access modifier'larını private final olarak tanımlayarak yapabiliriz.

```java
public class UserService {

  private final UserRepository userRepository;
  private final RabbitTemplate rabbitTemplate;

  @Autowired
  public UserService(UserRepository userRepository, RabbitTemplate
    this.userRepository = userRepository;
    this.rabbitTemplate = rabbitTemplate;
    }
}
```

Diğer bir yöntem ise Setter Dependency Injection yöntemidir. Bu yöntemde Sınıf yapımızda kullandığımız gibi setter metodunu kullanıyoruz. Private olarak tanımlandığımız servise dışardan aldığımız bir servisi inject ediyoruz.

```java
public class ChannelServiceImpl implements ChannelService {
  private UserService userService;

  @Autowired
  public void setUserService(UserService userService) {
    this.userService = userService;
  }
}
```

3. yöntem ise Field Dependency Injection yöntemidir.

```java
public class AuthController {
  @Autowired
  private AuthenticationManager authenticationManager;
  @Autowired
  private JwtTokenUtil jwtTokenUtil;
  @Autowired
  private JwtUserDetailsService userDetailsService;
}
```

Singleton tasarım deseni, bellekte istenilen nesnenin yalnızca tek bir örneğinin bulunmasını sağlar. Java tabanlı bean konfigürasyonunda bir bean'i singleton olarak tanımlamak için, aşağıdaki tanımlamayı yaparız. Uygulamada işlerimizi hallederken userService nesnesini bir kere çağırmamız bize yeterli olacaktır.

```java
@Configuration
public class AppConfiguration {

    @Bean
    @Scope("singleton") // default scope
    public UserService userService(){
        return new UserService();
    }
}
```

Spring, aşağıdaki iki yaklaşımı kullanarak bean nesnelerinin oluşturulması için factory tasarım modelini kullanır.

Spring BeanFactory Container: – Dependency Injection (Bağımlılık enjeksiyonu) için temel desteği sağlayan, Spring framework'te bulunan en basit container(kapsayıcı) yapısıdır. Bu container ile çalışmak için org.springframework.beans.factory.BeanFactory arayüzünü kullanıyoruz.

Spring ApplicationContext Container: Spring container'da bulunan ve ekstra işlevsellik ekleyen başka bir container'dır. Bu işlevler, metin mesajlarını bir özellikler dosyasından çözme ve uygulama olaylarını kullanıcılara yayınlama yeteneğini içerir. Bu kapsayıcı ile çalışmak için aşağıdaki arayüzü kullanıyoruz. Bu container ile çalışmak için org.springframework.context.ApplicationContext arayüzünü kullanıyoruz.

```
package net.javaguides.spring.ioc;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {
    public static void main(String[] args) {
        ApplicationContext  context = new AnnotationConfigApplicationContext(AppConfig.class);
        context.close();
    }
}
```

Proxy tasarım modelinde, başka bir sınıfın işlevselliğini temsil etmek için bir sınıf kullanılır. Yapısal desene bir örnektir. Burada, işlevselliğini dış dünyaya arayüz ile aktarmak için orijinal bir nesneye sahip bir nesne yaratılır.

Bir proxy oluşturmak için, sınıfımızla aynı arayüzü uygulayan ve sınıfa bir referans içeren bir nesne yaratıyoruz.Daha sonra sınıf yerine proxy'yi kullanabiliriz. Spring'de, bean'ler, alttaki bean'e erişimi kontrol etmek için temsil edilir. 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 notu ekliyoruz. Bu ek açıklama, Spring'e oluşturma yöntemimizi atomik olarak yürütmesi talimatını verir. Bir proxy olmadan Spring, BookRepository çekirdeğimize erişimi kontrol edemez ve işlem tutarlılığını sağlayamaz.

BookManager create metodunu çağırdığımızda çıktıyı görebiliriz:

```
com.baeldung.patterns.proxy.BookRepository$$EnhancerBySpringCGLIB$$3dc2b55c
```

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

İşin arkasında Spring, BookRepository nesnemizi EnhancerBySpringCGLIB nesnesi olarak içine sardı. Böylece Spring, BookRepository nesnemize erişimi kontrol eder (işlem tutarlılığını sağlar).

Template method (Şablon yöntemi) algoritmanın yapısını tanımlar ve alt sınıfların farklı uygulamalar sağlamasına izin vererek bir veya daha fazla adımda küçük değişikliklere izin verir. Veritabanı sorgusu durumuna göre bir şablon yaratabiliriz.

```
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);
}
```

executeQuery metodu yerine, bu yürütme yöntemine sonuçları işlemek için bir sorgu dizesi ve bir geri arama yöntemi sağlayabiliriz. İlk olarak, bir Results nesnesini alan ve onu T tipi bir nesneye eşleyen geri arama yöntemini oluşturuyoruz:

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

Ardından, bu geri çağırmayı 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 geri çağırma mekanizması, tam olarak Spring'in JdbcTemplate sınıfıyla kullandığı yaklaşımdır. JdbcTemplate sınıfı, String 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 tipi bir etki alanı nesnesine dönüştürür:

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

Spring, daha spesifik geri arama arabirimleri oluşturarak standart kodu daha da azaltır. Örneğin, RowMapper arayüzü, tek bir SQL verisi satırını T tipi bir etki alanı nesnesine dönüştürmek için kullanılır.

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

RowMapper arabirimini beklenen ResultSetExtractor'a uyarlamak için Spring, RowMapperResultSetExtractor sınıfını 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...
}
```

Satırların yinelenmesi de dahil olmak üzere tüm bir ResultSet nesnesinin dönüştürülmesi için mantık sağlamak yerine, tek bir satırın nasıl dönüştürüleceğine ilişkin mantık 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, JdbcTemplate'i kullanarak bir veritabanını sorgulayabilir ve ortaya çıkan her satırı eşleyebiliriz:

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

**SOA - Web Service - Restful Service - HTTP methods kavramlarını örneklerle açıklayınız.**

Web servis HTTP protokolü üzerinden diğer sistemlere/cihazlara hizmet veren yapılardır. Http ise bilginin sunucudan kullanıcılara nasıl ve ne şekilde aktarılacağını gösteren protokoldür. Http kullanıcıların aygıtları ve sunucu arasındaki veri alışverişinin kurallarını belirler. Kullanıcılar girmek istediği sitenin adresini tarayıcı çubuğunuza yazarlar, bunu yazdıktan sonra HTTP yardımı ile siteye bir bağlantı isteği gider. Bu bağlantı isteğini kabul eden sitenin sunucusu ile bağlantı kurulur ve kullanıcı internet sitesine girmiş olur. Eğer sunucudan bir cevap gelmezse kullanıcı bir çeşit hata uyarısı alır.


HTTP protokolünde, sunucuya gönderilen istek içinde yapılması istenilen işlem tiplerine Http metotları adı verilir. Bu metotlar kullanıcının ne yapmak istediğini ne amaç ile sunucuya bağlanmak istediğini belirtmek için kullanılır.

GET: Çağrılan adresten bilgi almak için kullanılır.

HEAD: Çağrılan adresten cevap başlıkları (response header) ile bilgi almak amacıyla kullanılır. GET ile farkı cevap bölümü (response body) olmamasıdır ve tüm bilgi başlıktadır.

POST: Çağrılan adrese bilgi göndermek için kullanılır.

PUT: Sunucudaki bir kaynağı güncellemek için kullanılır. Bu istekler de genellikle üzerlerinde değiştirilmek istenen bilgiyi taşırlar.

PATCH: Sunucudaki bir kaynağı değiştirmek için kullanılır. Put ile arasındaki fark ise Put sunucudaki kaynağı yeni bir kaynak ile değiştirmek için kullanılır iken, Patch bu kaynağında bir kısmını değiştirmeye yarar.

DELETE: Sunucu tarafında bir kaynağı silmek amacı ile kullanılır.

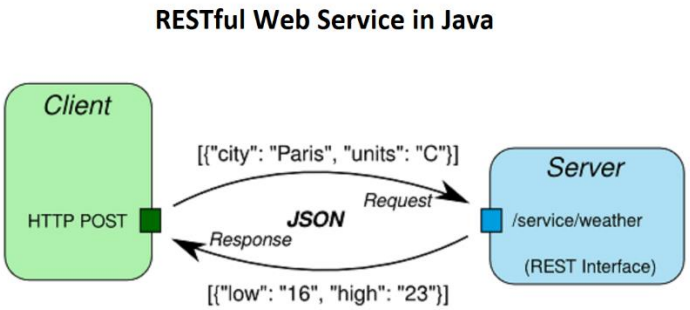
OPTIONS: Çağrılan adres (kaynak) ile ilgili desteklenen method'lar alınır.

TRACE: Sucuya yapılan isteği aynen geri almak için yapılır. Sunucuyu test etme ve inceleme amacıyla yapılır.

CONNECT: HTTPS isteklerinin HTTP içinden iletilmesi için kullanılır.

Web servislerinin temel özelliği, uygulamaların çeşitli dillerde yazılabilmesi ve istemciler ve sunucular arasındaki bir web servisi aracılığıyla birbirleriyle veri alışverişi yaparak iletişim kurabilmeleridir. Temel anlamda, REST ve SOAP olmak üzere 2 çeşit web service vardır. İkisi arasındaki en önemli fark ise REST ile xml, json, metin, html gibi istediğiniz türde çıktılar alıp gönderebilirken, SOAP Web Service ile sadece xml veriler ile işlem yapabiliriz. Restful web servisleri ise, REST mimarisi temel alınarak geliştirilmiş oldukça hafif, genişletilebilir ve basit servislerdir. Restful servislerin amacı client-server arasındaki veri akışını platform bağımsız olarak gerçekleştirebilmek ve veri akışını en az yükle sağlayabilmektir. REST

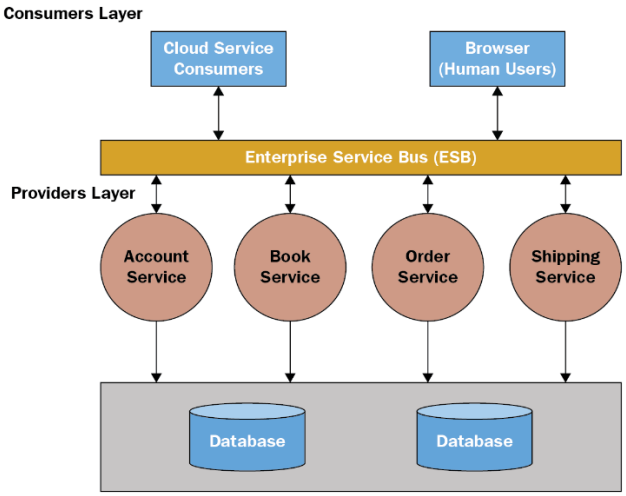
servisler URI ile ilgili metoda HTTP üzerinden istekte bulunur. GET, POST, PUT, DELETE gibi HTTP metotları ile işlemler gerçekleştirilebilmektedir.




Restful servisleri response tipi olarak JSON, HTML, XML gibi birçok formatta çalışabilirler. Yapısının az yer kaplaması ve daha kullanışlı olmasından dolayı response tipi olarak genellikle JSON kullanılmaktadır. Resimde görüldüğü üzere Client tarafından bazı bilgiler yollanıp ilgili şehrin Sıcaklık bilgisi istenmiştir. İlgili serviste cevabı kullanıcıya döndürmüştür. HTTP protokolü üzerinden bir POST işlemi yapılmıştır. Bilgiler Json formatında yollanıp Json formatında alınmıştır.

SOA (Service Oriented Architecture) temel olarak her hizmetin farklı birimler tarafından birbirinden bağımsız olarak çalışmasını ifade eder. Servis olarak adlandırdığımız, gevşek bağlı (loosely coupled) ve iri taneli (coarse grained) yapıdaki bileşenlere dayalı dağıtık sistemlerin geliştirilmesi için kullandığımız bir mimari yaklaşımdır. Soa içindeki her hizmet, eksiksiz, bağımsız bir şekilde çalışmalıdır.

Örneğin bir banka uygulaması düşünelim, bir müşteri kredisinin kontrolünü, aylık kredilerinin ödemesinin hesaplanması, kredi kartı ödemelerinin hesaplanması veya kredi başvurularına ait işlemlerini takip etmesi için kapsamlı bir mimari gerekmektedir. Bu işlemler yapıldığı sırada hizmetler birbirileri arasında sadece istenen verileri elde etmeyi ya da göndermeyi hedefler, diğer birimler ile ilgili başka işlere odaklanmazlar. Hizmetler, verileri okuma veya değiştirme isteği göndermek için SOAP (basit nesne erişimi protokolü), HTTP veya JSON/HTTP gibi standart ağ protokolleri kullanılarak ortaya çıkarılır.




ESB (Enterprise Service Bus) kurumsal hizmet veri yolu, merkezi bir bileşenin arka uç sistemlerle entegrasyonu gerçekleştirip bunların hizmet arayüzleri olarak kullanabilmesini sağlayan bir modeldir.

Veri modellerinin çevirisini, derin bağlanırlığı, yönlendirmeyi ve birden çok isteğin potansiyel bileşimini gerçekleştirir ve yeni uygulamalar tarafından yeniden kullanılması için tek bir hizmet arayüzü olarak sunar.

## Kaynaklar:


- ✓ Digitalocean
- ✓ Tuğrulbayrak medium
- ✓ Programiz
- ✓ javaguides
- ✓ tutorialspoint
- ✓ devnot
- ✓ baeldung
- ✓ interviewbit
- ✓ javatpoint