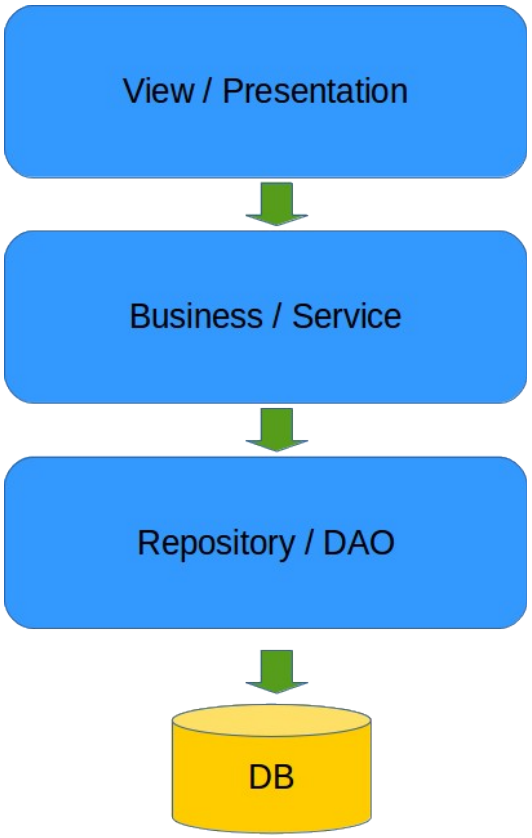
ServiceLocator'dan Inversion of Control'e Kısa bir Yolculuk

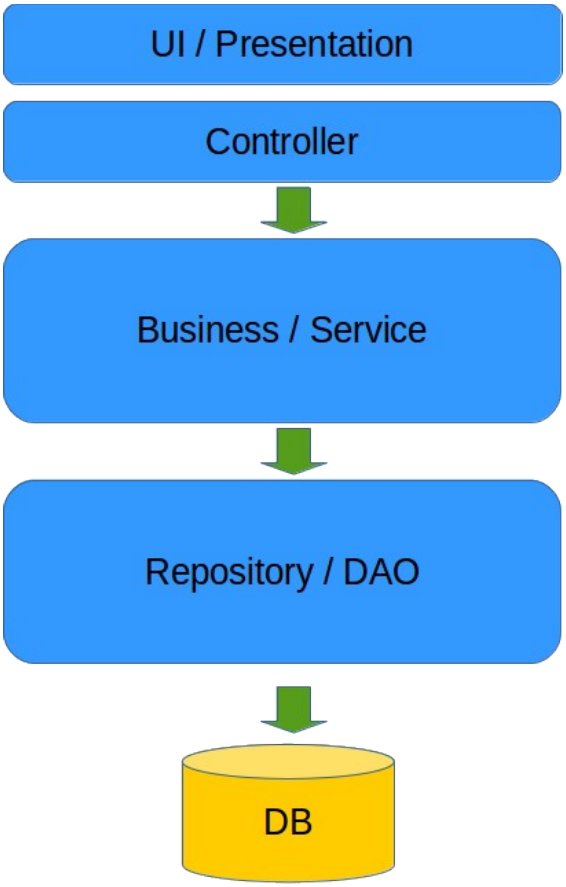
Kenan Sevindik, Harezmi Bilişim Çözümleri

JavaEE'nin, yani Enterprise (Kurumsal) Java'nın daha J2EE olarak anıldığı ilk dönemlerde yazılımcılar, geliştirdikleri kurumsal uygulamaların mimarileri için Sun Microsystems'in yayımladığı J2EE Blueprints dokümanlarında önerilen katmanlı mimari modeli baz almaktaydılar. Aslında katmanlı mimari modelin geçmişi çok daha eskilere, David Parnas'ın sistem mimarisi ve modülarizasyon ile ilgili makalelerine kadar uzanmaktadır. Ancak o dönem, geliştiriciler için klasik istemci-sunucu mimarilerinden sonra katmanlı mimari model, sistemi daha modüler bir yapıya oturtmak, anlaşılırlığını ve bakım idamesini kolaylaştırmak açısından büyük bir yenilik olarak algılanmıştır.



Üç katmanlı veya N katmanlı mimari olarak bilinen bu modelde her bir katman spesifik belirli bir göreve odaklanmaktadır ve sadece bir altındaki katman ile iletişim kurmaktadır. Her katman bir üstteki katmana, kullanması için ihtiyaç duyduğu servisleri belirli bir kontrat üzerinden sunmaktadır. Böylece sistemin genel olarak anlaşılması ve üzerinde çalışılması daha kolay olmaktadır. Sistemin katmanları ekip içerisinde paylaşılarak farklı gruplar tarafından eş zamanlı

olarak geliştirilebilmektedir. “Program to interface” yaklaşımı uygulandığı takdirde bu kontratlar sınıf arayüzleri olarak tanımlanmakta ve her bir katman bir altındaki katmanın gerçekleştirim detaylarından bağımsız biçimde geliştirilebilmekte ve test edilebilmektedir. Ayrıca alttaki katman üstteki katmanı etkilemeden farklı gerçekleştirimlere sahip olabilmektedir.



Katmanlı mimarinin en yaygın biçimi olan üç katmanlı mimari model, görünüm, servis ve veri erişim katmanlarından oluşmaktadır. Görünüm katmanı da kendi içerisinde zaman zaman sunum ve kotrol olarak ayrıştırılabilmektedir. Bu durumda katmanlı mimari model dört katmandan oluşmaktadır. Bu nedenle de çok katmanlı veya N katmanlı mimari model şeklinde anılabilmektedir. Kurumsal Java uygulamaları ilk dönemden itibaren ağırlıklı olarak web platformunda çalışacak biçimde geliştirildiklerinden sunum ve kontrol katmanlarında JSP ve Servlet teknolojileri ile geliştirilmiş nesneler, servis katmanında Session EJB nesneleri, veri erişim katmanında ise Entity Bean veya sıradan Java nesneleri şeklinde geliştirilmiş DAO nesneleri yer almaktaydı. Her katman da bir üstteki katmana, değişmez, sabit bir kontrat sunmayı hedeflediğinden, sunum katmanının servis katmanındaki Session EJB'lerine bağımlı olmaması için EJB teknolojisi “Business Delegate” örüntüsü ile bir arayüz arkasında gizlenirdi. Benzer biçimde DAO sınıfları için de ayrı arayüzler tanımlanarak farklı DAO gerçekleştirimlerinin servis katmanı tarafından kullanılabilmesi sağlanırdı.

Ayrıca bu yaklaşım katmanların da birbirlerinden bağımsız biçimde test edilebilmesini mümkün kılmakta ve kolaylaştırmaktaydı.

Katmanlı mimari modeldeki nesnelerin diğer bir özelliği ise çoğunlukla “stateless” olmalarıdır. Yani nesneler iki farklı metot çağrısı arasında herhangi bir durum bilgisi saklamazlar ve her bir metot çağrısında senaryo veya iş mantığı kendi içerisinde başlayıp sona erer. Dolayısı ile sunum katmanındaki Servlet nesneleri, servis katmanındaki Stateless Session EJB'ler ve DAO katmanındaki veri erişim nesneleri aynı anda birden fazla isteğe cevap verebilir. Başka bir ifade ile belirli bir Servlet, Session EJB veya DAO sınıfından tek bir nesne yaratılıp, sistemin bütün istekleri bu nesneler üzerinden karşılanabilir. Sistem genelinde bir sınıftan tek bir nesnenin olmasını sağlayan veya şart koşan örüntüye Singleton adı verilmektedir.

```
public interface AuthenticationService {
	public Authentication authenticate(String username, String password);
}
```

```
public interface UserDao {
	public User findByUsername(String username);
}
```

```
public class DefaultAuthenticationService implements AuthenticationService {

	private UserDao userDao;

	@Override
	public Authentication authenticate(String username, String password) {
		User user = userDao.findByUsername(username);
		if (user != null && user.getPassword().equals(password)) {
			return new Authentication(username, user.getRoles());
		}
		throw new AuthenticationFailedException("Cannot authenticate
user :" + username);
	}

}
```

Yukarıdaki örnekte AuthenticationService ve UserDao isimli iki arayüz ve DefaultAuthenticationService isimli bir gerçekleştirim mevcuttur. DefaultAuthenticationService sınıfında kimliklendirme (authentication) işlemini gerçekleştirirken kullanıcı bilgisini elde etmek için UserDao arayüzünden bir nesneye ihtiyaç duyulmaktadır. DefaultAuthenticationService UserDao arayüzünün findByUsername metodunu kullanarak User nesnesini elde etmeye çalışır ve girilen username'e sahip mevcut bir User var ise bunun şifresi ile sunum katmanından gelen şifreyi

karşılaştırarak kimliklendirme işleminin sonucunu istemci koduna döndürür. İşte tam bu noktada her bir katmandaki nesnenin bir alttaki veya kendi katmanındaki diğer singleton nesnelere nasıl erişeceği ve istekleri bunlara nasıl göndereceği sorusu ortaya çıkmaktadır.

```
public class DefaultAuthenticationService implements AuthenticationService {

      private UserDao userDao = new JdbcUserDao(new DriverManagerDataSource());

      ...

}
```

En basit ve ilk akla gelen çözüm yukarıdaki gibi UserDao arayüzünden bir gerçekleştirimi, örneğin JdbcUserDao, DefaultAuthenticationService sınıfı içerisinde yaratarak UserDao nesnesi elde etmektir. Ancak bu açıklık-kapalılık (open-closed) tasarım prensibini (OCP) ihlal eden bir çözüm olacaktır. Açıklık kapalılık prensibi bize, yeni gereksinimler ortaya çıktığında bu gereksinimleri karşılamak için sınıfların içerisinde değişikliğe gitmeyip (closed for modification), mevcut kod üzerinde genişlemeye giderek (open for extension) problemi çözmeyi tavsiye eder. Başka bir ifade ile sınıflar değişikliklere kapalı, yalnızca genişlemeye açık olmalıdır. Bu prensibe dayalı olarak concrete bir sınıf sadece arayüz veya soyut sınıflara bağımlı olmalıdır. Concrete sınıflar diğer concrete sınıflara bağımlı olmamalıdır, onları bilmemelidir. Örneğin, DefaultAuthenticationService'ın kullanıcı bilgilerine, ilişkisel bir veritabanı yerine LDAP repository'den erişmesi istenirse, DefaultAuthenticationService sınıfı içerisinde LdapUserDao gerçekleştiriminden bir nesne yaratılmalıdır. Bunun içinde DefaultAuthenticationService sınıfının değiştirilmesi ve yeniden derlenmesi gerekir. Bu da açıklık kapalılık prensibinin ihlali demektir. Değişen ve yeniden derlenen her sınıf tekrardan testlere tabi tutulmalıdır. Gerçekleşen değişikliklerden sonra da bu sınıfın daha önceden olduğu gibi hatasız çalıştığı bu testlere tabi tutulmadan garanti edilemez.

Bu örnekte ayrıca tek sorumluluk (single responsibility) prensibi (SRP) de ihlal edilmektedir. AuthenticationService arayüzü kimliklendirme davranışı tanımlamaktadır. DefaultAuthenticationService'ın asli sorumluluğu da bu davranışı gerçekleştirmektir. Ancak DefaultAuthenticationService içerisinde aynı zamanda UserDao arayüzünden bir nesne de yaratılmaktadır. Oysa nesne yaratma ayrı bir sorumluluktur ve DefaultAuthenticationService'ın asli görevi değildir ve olmamalıdır. Aşağıdaki örnekte UserDao yaratma işlemi ayrı bir metot içerisine çekilerek encapsule edilmiştir.

```
public class DefaultAuthenticationService implements AuthenticationService {

	private UserDao userDao = createUserDao();

	private UserDao createUserDao() {
		return new JdbcUserDao(new JdbcDataSource());
	}

	...

}
```

Buradaki createUserDao() metodu nesne yaratma işleminden sorumludur. Bu tür metotlara Factory Method adı verilmektedir. Bu örnekte her ne kadar Factory Method örüntüsü çözüme sınırlı bir biçimde dahil edilmiş olsa bile yukarıda bahsedilen problemler için henüz herhangi bir iyileşme söz konusu değildir. Bir an için bu OCP ve SRP gibi iki temel prensibin ihlalini göz ardı ederek DefaultAuthenticationService'i bazı ortamlarda kullanıcı bilgilerini bir property dosyasından yükleyip dönen InMemoryUserDao ile, bazı ortamlarda kullanıcı bilgilerini JDBC API üzerinden dönen JdbcUserDao ile, diğer bazı ortamlarda ise JNDI üzerinden dönen LdapUserDao ile çalışacak biçimde değiştirelim.

```
public class DefaultAuthenticationService implements AuthenticationService {

	private UserDao userDao = createUserDao();

	private UserDao createUserDao() {
		String targetPlatform = System.getProperty("targetPlatform");
		if("dev".equals(targetPlatform)) {
			return new InMemoryUserDao();
		} else if("test".equals(targetPlatform)) {
			return new JdbcUserDao(new JdbcDataSource());
		} else {
			return new LdapUserDao();
		}
	}

	...

}
```

Bunu gerçekleştirmenin en basit yolu yukarıdaki gibi bir sistem veya ortam değişkeninin mevcut değerine göre InMemoryUserDao, JdbcUserDao veya LdapUserDao gerçekleştirimlerinden uygun olanını kullanarak bir UserDao nesnesi yaratmak olacaktır. Ancak bu yaklaşım UserDao'nin başka bir ortam için daha farklı bir gerçekleştirimini kullanmak istediğimizde DefaultAuthenticationService'ın içerisinin değiştirilmesine neden olacağı için çözümün ilk halinden

çok da farklı olmayacaktır. Sonuç olarak, hem açıklık kapalılık, hem de tek sorumluluk prensiplerini gözetirsek en doğru yaklaşım UserDao nesnesinin yaratılması işinin DefaultAuthenticationService dışında bir yerde gerçekleştirilmesi olacaktır. Bu durumda DefaultAuthenticationService mevcut ortam için uygun UserDao nesnesini başka bir yerden elde etmeye çalışmalıdır. Bunun için J2EE ile programlamanın ilk dönemlerindeki en yaygın yaklaşım ServiceLocator örüntüsünün kullanılması olmuştur.

```
public class ServiceLocator {

	public static final ServiceLocator INSTANCE = new ServiceLocator();

	private AuthenticationService authenticationService;
	private UserDao userDao;
	private DataSource dataSource;

	private ServiceLocator() {
		try {
			String targetPlatform = System.getProperty("targetPlatform");
			InputStream is = this.getClass().getClassLoader()
					.getResourceAsStream(
						"service." + targetPlatform + ".properties");
			Properties properties = new Properties();
			properties.load(is);
			authenticationService = (AuthenticationService) Class
					.forName(properties.getProperty(
							"authenticationService")).newInstance();
			userDao = (UserDao) Class.forName(
					properties.getProperty("userDao")).newInstance();
			dataSource = (DataSource) Class.forName(
				properties.getProperty("dataSource")).newInstance();
		} catch (Exception ex) {
			throw new ServiceLoadFailedException(
					"Cannot load services", ex);
		}
	}

	public AuthenticationService getAuthenticationService() {
		return authenticationService;
	}

	public UserDao getUserDao() {
		return userDao;
	}

	public DataSource getDataSource() {
		return dataSource;
	}
}
```

Bu örüntüde ServiceLocator nesnesi farklı ortamlar için farklı gerçekleştirimleri yaratıp, ilgili nesnelere hangi bağımlılıkları istiyorlarsa sunma görevini üstlenmektedir. Farklı her ortam için yaratılacak servis gerçekleştirimleri service.dev.properties, service.test.properties,

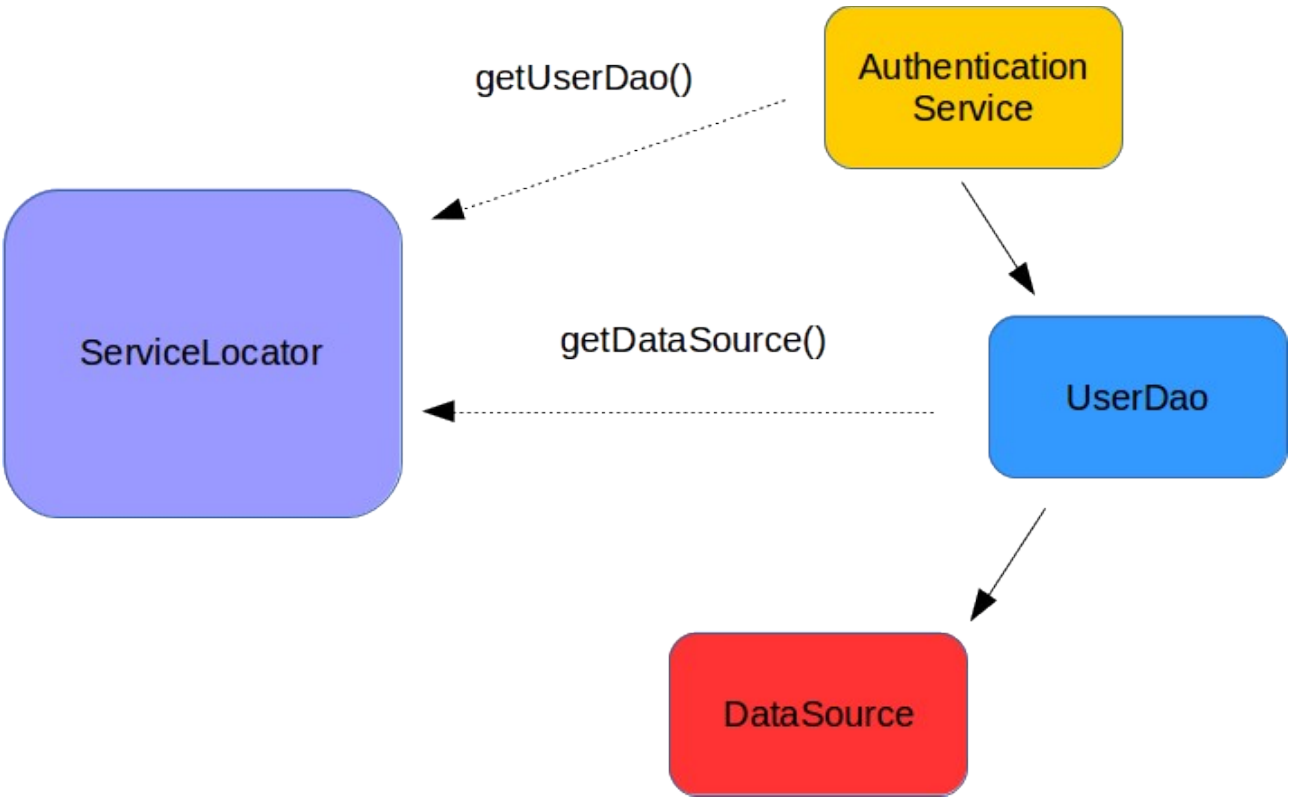
service.prod.properties gibi property dosyaları içerisinde tanımlanmaktadır.

```
authenticationService=com.javaegitimleri.DefaultAuthenticationService
userDao=com.javaegitimleri.JdbcUserDao
dataSource=com.javaegitimleri.JdbcDataSource
```

Property dosyasındaki her bir satır ayrı bir servis tanımıdır. ServiceLocator'dan herhangi bir bağımlılığın veya başka bir ifade ile servisin talep edilmesi ise bu servisi diğerlerinden benzersiz biçimde ayrıştırmayı sağlayacak bir niteleyici vasıtası ile olmaktadır. Bu niteleyici, servisin sahip olduğu bir arayüz veya her servise atanan -servis adı gibi- benzersiz bir String değer olabilir. Örnekteki Property dosyasındaki key değerleri de buradaki servis isimlerine karşılık gelmektedir.

```
public class DefaultAuthenticationService implements AuthenticationService {

	private UserDao userDao = ServiceLocator.INSTANCE.getUserDao();

	...
}
```

```
public class JdbcUserDao implements UserDao {

	private DataSource dataSource = ServiceLocator.INSTANCE.getDataSource();

	...
}
```



Son aşamada ServiceLocator ortama göre yaratılan servis nesnelerini talebe göre dönmektedir. Görüldüğü üzere ServiceLocator sınıfından da sistem genelinde bir tane yaratılıp bu nesnenin kullanılması yeterli olmaktadır. Dolayısı ile ServiceLocator'da singleton bir nesnedir.

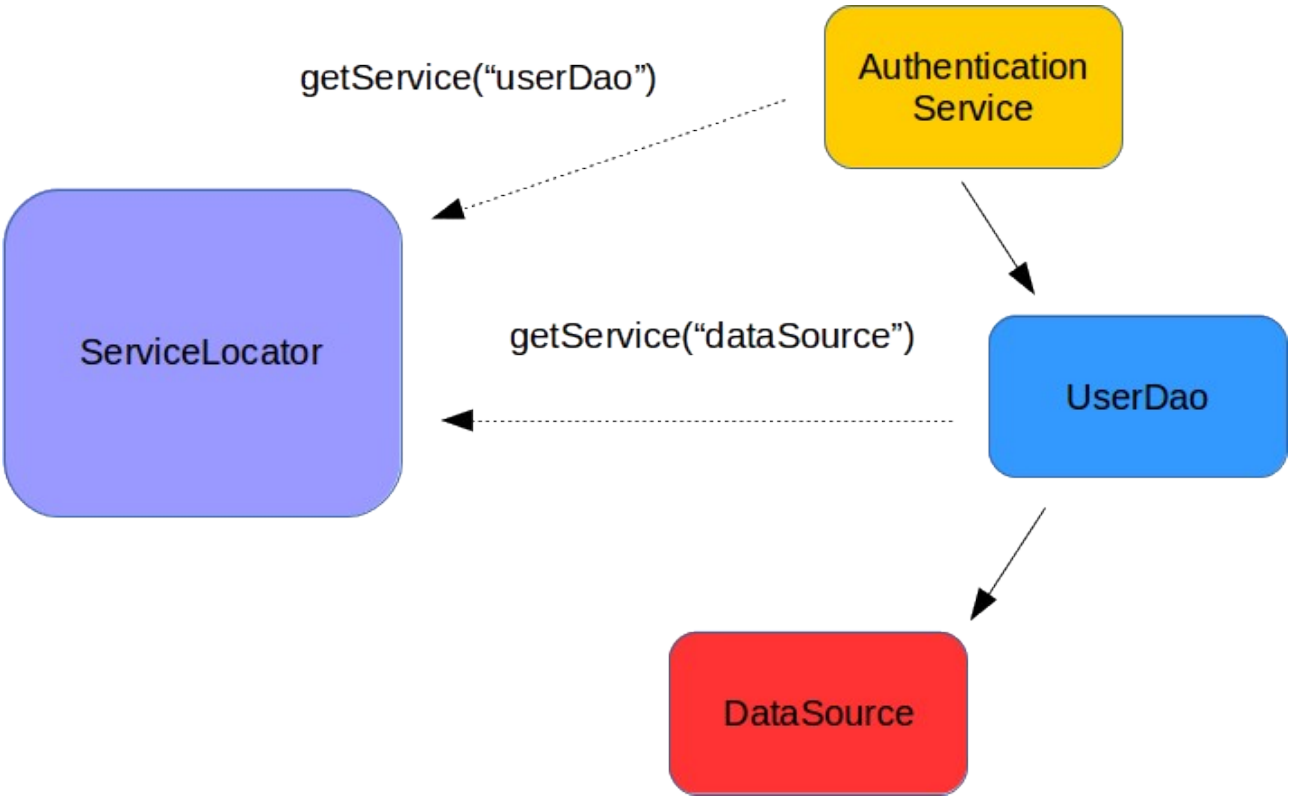
ServiceLocator'daki servisleri yaratma ve yönetme işi biraz daha genel kullanıma uygun hale getirilirse getAuthenticationService(), getUserDao(), getDataSource() gibi spesifik metotlar yerine getService(String serviceName) gibi tek bir metot vasıtası ile uygulamadaki olası bütün servislere erişim sağlanabilir.

```
public class ServiceLocator {

	public static final ServiceLocator INSTANCE = new ServiceLocator();

	private Map<String, Object> serviceMap = new HashMap<>();

	private ServiceLocator() {
		try {
			String targetPlatform = System.getProperty("targetPlatform");
			InputStream is = this.getClass().getClassLoader()
					.getResourceAsStream("service." + targetPlatform
+ ".properties");
			Properties properties = new Properties();
			properties.load(is);
			for (Object serviceName : properties.keySet()) {
				Object service =
Class.forName(properties.getProperty(serviceName.toString())).newInstance();
```

```
                        serviceMap.put(serviceName.toString(), service);
                  }
            } catch (Exception ex) {
                  throw new ServiceLoadFailedException("Cannot load services",
ex);
            }
      }

      @SuppressWarnings("unchecked")
      public <T> T getService(String serviceName) {
            return (T) serviceMap.get(serviceName);
      }
}
```

```
public class DefaultAuthenticationService implements AuthenticationService {
      private UserDao userDao = ServiceLocator.INSTANCE.getService("userDao");
      ...

}
```

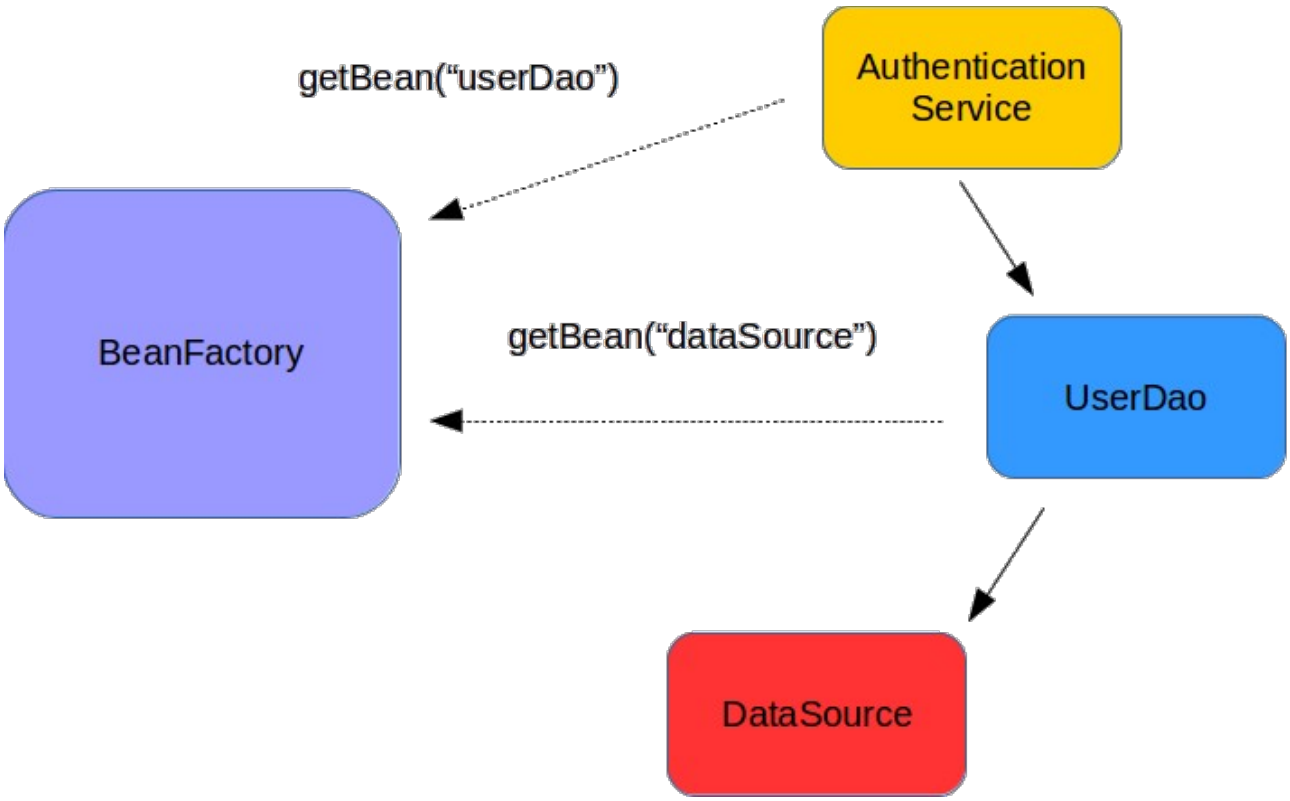
```
public class JdbcUserDao implements UserDao {
      private DataSource dataSource =
ServiceLocator.INSTANCE.getService("dataSource");

      ...
}
```



Zaman içerisinde ServiceLocator üzerinden ilgili nesnelerin yaratılıp sunulmasında gereksinimler daha da ileriye gitmiştir. Örneğin, bazı sınıflardan uygulama genelinde tek bir nesne

(singleton) yeterli olmasına rağmen, bazılarında her erişim için farklı bir nesnenin yaratılması veya web isteği veya oturumu boyunca (request/session scope) aynı nesnenin kullanılması, ancak farklı istekler ve oturumlar için farklı nesnelerin kullanılması, servislerin yaratılmasının uygulamanın başlatılması anında (eager initialization) değil, sadece gerçekten ihtiyaç olduğu anda (lazy initialization) gerçekleştirilmesi ve buna benzer pek çok gereksinim de o dönemde kurumsal uygulama geliştiricilerin karşısına çıkmıştır.

İşte tam da bu noktada EJB 2.0 ve 2.1 merkezinde ortaya çıkan kurumsal Java uygulamalarının geliştirilmesindeki zorlukları aşmak ve yukarıdakilere benzer pek çok gereksinimi karşılamak için Spring Application Framework ortaya çıkmıştır. Spring Application Framework'ün ortaya çıkışında Dependency Injection veya diğer ismi ile Inversion of Control örüntüsü üzerinden nesne yaratma ve bağımlılıkların karşılanması konusu temel teşkil etmiştir. Bu örüntüde nesne veya servis kendisine ihtiyaç duyan nesne içerisinde değil, onun dışında başka bir yerde yaratılır ve kendisine ihtiyaç duyan nesnenin içerisine bir property olarak setter metot vasıtası ile veya constructor parametresi şeklinde “enjekte” edilir. Böylece nesnelerin yaratılması ve bağımlılıkların yönetilmesi nesnelerin kendilerinden çıkarak onların dışında bir ortama geçmiştir. Bu örüntü ile birlikte nesnelerin ServiceLocator nesnesine doğru olan bağımlılıkları da ortadan kalmaktadır. Çünkü nesnelerin ihtiyaç duydukları bağımlılıklar kendilerine bu yeni ortam tarafından enjekte edilmektedir. Bu ortam, yani Spring Container veya başka bir ismi ile ApplicationContext, nesne veya servislerin, Spring terminolojisi ile söylersek Spring managed bean'ların, yaratılması ve diğer ihtiyaç duyan ilgili bean'lara sunulmasından görevlidir. Genel olarak Spring Container temel kabiliyet olarak, nesnelerin yaratılması, yönetilmesi ve bağımlılıkların karşılanması problemine sistematik bir yol sunmaktadır. Spring Container'ın bu görevini üstlenen bölüm BeanFactory arayüzüdür. BeanFactory arayüzü burada tam olarak ServiceLocator'ın yaptığı işi üstlenmiştir. BeanFactory, Spring tarafından yönetilen bean'ları yaratmak ve aralarındaki bağımlılıkları yönetmekten sorumludur.



ServiceLocator'da olduğu gibi benzer biçimde BeanFactory'de de Spring bean'lara isimleri veya tipleri ile erişilebilmektedir. Spring ApplicationContext ise BeanFactory arayüzüne sahip olmakla beraber, daha gelişmiş ve pek çok ilave kabiliyeti sunan bir yapıdır. ApplicationContext ile BeanFactory ilişkisi Decorator örüntüsü üzerinden kurgulanmıştır. ApplicationContext, BeanFactory arayüzüne sahip olmakta birlikte bean yaratma ve bağımlılıkları enjekte etme işini gerçekleştiren bir BeanFactory nesnesini de kendi içerisinde barındırmaktadır. ServiceLocator örüntüsünde olduğu gibi BeanFactory'nin kullanımında da nesnelerin hangi sınıflardan yaratılacağı, hangi property'lere bağımlık olarak enjekte edilecekleri uygulama geliştiriciler tarafından “metadata” olarak belirtilmelidir. Bean konfigürasyon verisi olarak da adlandırılan bu metadata, XML, annotasyon veya Java sınıfları şeklinde tanımlanabilir. Spring, tarihsel süreç içerisinde format olarak XML ile başlamış, ilerleyen zamanlarda her üç formatı da aynı anda destekler bir noktaya ulaşmıştır.

```
<?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.xsd">

      <bean id="authenticationService"
class="com.javaegitimleri.DefaultAuthenticationService">
            <property name="userDao" ref="userDao"/>
      </bean>

      <bean id="userDao" class="com.javaegitimleri.JdbcUserDao">
            <constructor-arg ref="dataSource"/>
      </bean>

      <bean id="dataSource" class="com.javaegitimleri.JdbcDataSource"/>

</beans>
```

```
ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:/applicationContext.xml");

AuthenticationService authenticationService = applicationContext.getBean(
                        "authenticationService",AuthenticationService.class);

authenticationService.authenticate("user1", "secret");
```

Yukarıdaki örnekte XML formatında bir Spring bean konfigürasyon dosyası oluşturulmuş ve içerisinde bean tanımları yapılmıştır. Ardından Spring'in ClasspathXmlApplicationContext gerçekleştirimi ile de bu konfigürasyon dosyası yüklenerek bir ApplicationContext yaratılmıştır. Yaratılan ApplicationContext yani Spring Container içerisinden de getBean() metodu ile AuthenticationService bean'ine erişilmiş ve ilgili iş mantığı – kimliklendirme işlemi - çalıştırılmıştır. Spring bean'ları sıradan Java nesneleridir. Herhangi bir özel sınıftan veya arayüzden türemeleri zorunlu değildir. Her bir bean tanımının bir ismi vardır. Bean'lere diğer bean tanımlarından bu isimle referans verilebilir veya uygulama içerisinden bu isimle erişilebilir.

Spring'in ilk çıktığı dönem Java 5 öncesine rastladığı için o zamanlar Java anotasyonları mevcut değildi, ve XML en geçerli metadata oluşturma formatı idi. Java 5 ile birlikte anotasyonlar desteklenmeye başlayınca, Java uygulamalarında metadata tanımlama formatı olarak Java anotasyonları da kullanılmaya başlandı.

```java
@Service("authenticationService")
public class DefaultAuthenticationService implements AuthenticationService {

	private UserDao userDao;

	@Autowired
	public void setUserDao(UserDao userDao) {
		this.userDao = userDao;
	}

	...
}
```

```java
@Repository
public class JdbcUserDao implements UserDao {

	private DataSource dataSource;

	@Autowired
	public JdbcUserDao(DataSource dataSource) {
		super();
		this.dataSource = dataSource;
	}

	...
}
```

```java
@Component
public class JdbcDataSource implements DataSource {
     ...
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:context="http://www.springframework.org/schema/context"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
		http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

	<context:component-scan base-package="com.javaegitimleri"/>
</beans>
```

```java
ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:/applicationContext.xml");


AuthenticationService authenticationService = applicationContext.getBean(
                          "authenticationService",AuthenticationService.class);


authenticationService.authenticate("user1", "secret");
```

Yukarıdaki örnekte de görüldüğü üzere, Java sınıfları üzerinde kullanılan @Service, @Repository, @Component ve @Autowired gibi anotasyonlar ile bean tanımlama işlemi XML konfigürasyon dosyalarından Java sınıflarına doğru kaydırılmış oldu.

```
@Configuration
public class AppConfig {
	@Bean
	public AuthenticationService authenticationService(UserDao userDao) {
		DefaultAuthenticationService bean = new
DefaultAuthenticationService();
		bean.setUserDao(userDao);
		return bean;
	}

	@Bean
	public UserDao userDao(DataSource dataSource) {
		JdbcUserDao bean = new JdbcUserDao(dataSource);
		return bean;
	}

	@Bean
	public DataSource dataSource() {
		JdbcDataSource bean = new JdbcDataSource();
		return bean;
	}
}
```

```
AnnotationConfigApplicationContext applicationContext = new
AnnotationConfigApplicationContext(AppConfig.class);

AuthenticationService authenticationService = applicationContext.getBean(
                          "authenticationService",AuthenticationService.class);

authenticationService.authenticate("user1", "secret");
```

İlerleyen yıllar içerisinde bean tanımlarını ve ApplicationContext konfigürasyonunu tamamen Java sınıfları ile yapmak da mümkün hale geldi. Yukarıdaki örnekte XML konfigürasyon dosyasının bire bir karşılığı olan bir Java konfigürasyon sınıfı oluşturulmuş ve Spring ApplicationContext bu konfigürasyon sınıfı yüklenerek yaratılmıştır.

Sonuç olarak Spring Application Framework ile nesnelerin yönetilmesi ve bağımlılıkların karşılanması nesnelerin kendi sorumlulukları olmaktan çıkarak, Spring Container tarafından yürütülen bir iş haline dönüşmüştür. Spring Application Framework ise bu ve daha pek çok özelliği sayesinde kurumsal Java teknolojileri ile uygulama geliştiren yazılımcılar için vazgeçilmez bir iskelet framework olarak hayatımızda yerini almıştır.

Yazar Hakkında

1999 yılında ODTÜ Bilgisayar Mühendisliği'nden mezun olan Kenan Sevindik'in Java ile tanışıklığı 1998 yılına kadar uzanmaktadır. Yazar, Kurumsal Java teknolojilerini kullanarak uygulama geliştirme konusunda 19 yıldan fazla bir deneyime sahiptir. Java ile çalışmaya öğrencilik yıllarında, uzaktan eğitim programları için geliştirdiği Java Applet'lar ile başlamıştır. Spring Application Framework, Spring Security Framework, Hibernate Persistence Framework gibi çeşitli kurumsal Java teknoloji ve frameworkleri ile ilk çıktıkları dönemden bu yana çalışmalarını sürdürmektedir. Hali hazırda Harezmi Bilişim Çözümleri bünyesinde kurumsal yazılım geliştirme faaliyetlerine devam etmenin yanında, Java, OOP, Tasarım Örüntüleri, AOP, Spring, Spring Security, Hibernate gibi konularda eğitim ve danışmanlık hizmetleri de vermektedir. Teknoloji içerikli seminerlerde ve organizasyonlarda konuşmacı olarak da yer almaktadır. 2015 Şubat ayında Wiley Publishing tarafından yayımlanan "Beginning Spring" isimli kitabın yazarlarındandır. Teknik içerikli paylaşımlarına http://blog.harezmi.com.tr, düzenlediği eğitimlerle ilgili detaylı bilgiye ise http://www.java-egitimleri.com adreslerinden ulaşabilirsiniz.