Java ServiceLoader Nedir? SPI (Service Provider Interface) Nasıl Yazılır?

Rahman USTA

Java programlama dilinde yazılan standartların (spesifications) genişletilebilir kılınması için, `java.util.ServiceLoader` sınıfı Java 6'dan beri bulunmaktadır. `ServiceLoader` sayesinde yazılan spec'ler farklı uygulayıcılar tarafından karşılanabilmekte ve mevcut standart servis sistemine eklentiler yazılabilmektedir.

Örneğin bir servis sağlayıcısının `RandomServiceProvider` adında bize bir arayüz sunduğunu varsayalım;

```
public interface RandomServiceProvider {

    public Integer random(int start, int end);

}
```

`RandomServiceProvider` sınıfının bulunduğu projenin dizin yapısı aşağıdaki gibidir.

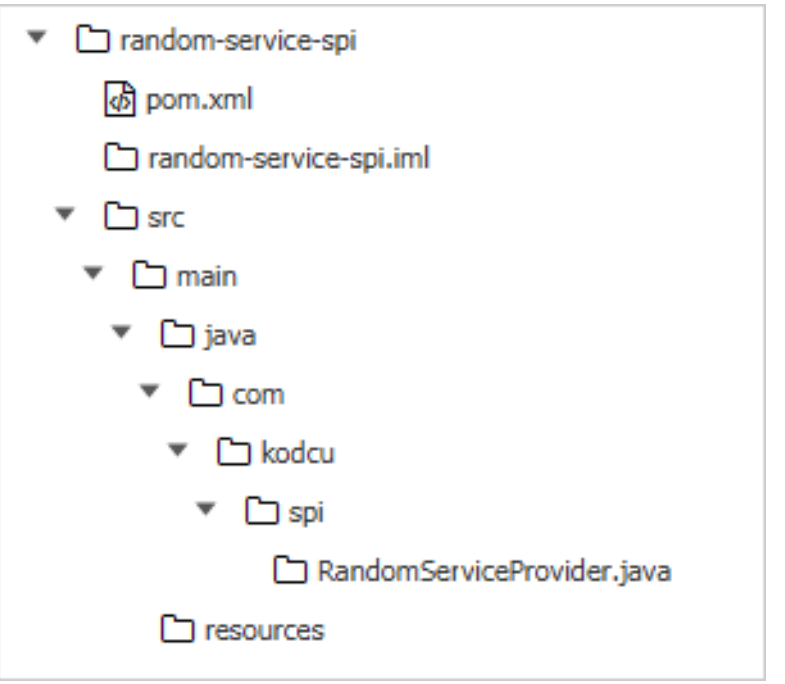


Figure 1. random-service-spi projesi

Bu servisin görevi adından da anlaşılacağı üzere verilen iki tam sayı arasında rastegele bir sayı üretmesidir. Servis sağlayıcı sınıfları genel olarak arayüz veya soyut sınıf türünden olmaktadır.

Spesifikasyon belirleyicileri, farklı uygulayıcılar için yukarıdakine benzer arayüzleri API uygulayıcılarına açmaktadır. Java standartlarının geliştirim sürecine bakıldığında, buna benzer bir süreç sürdürülmektedir. API uygulayıcıları ise bu arayüzlere dönük kendi implementasyonlarını geliştirmektedir.

Fakat iş sadece arayüzü tanımlamak değildir, bunun yanında bu sisteme standart uygulayıcıların ekler yapabilme imkanı olması gerekmektedir. `java.util.ServiceLoader` sınıfı bu noktada, servis uygulayıcılarına standart arayüz servislerine eklenti olarak eklenme imkanı sunmaktadır. Şimdi bu arayüzümüzü `ServiceLoader` sınıfı ile adım adım geliştirelim.

```
public interface RandomServiceProvider {

    public Integer random(int start, int end);

    public static RandomServiceProvider getProvider(String providerName) {
        return ...;
```

```
    }

    public static RandomServiceProvider getDefaultProvider() {
        return ...;
    }

}
```

`RandomServiceProvider` sınıfına `getProvider` ve `getDefaultProvider` adlarında iki statik metod ekledik. Bunlardan `getDefaultProvider()` çağrıldığında bu API'nin varsayılan imlpementasyonunun döndürülmesini istiyoruz. `getProvider(..)` olanla ise metoda sunulan sağlayıcı ismiyle `X` bir sağlayıcının döndürülmesini istiyoruz. İşte bu sağlayıcı sınıflar türünden uygulayıcı nesnelerin sisteme sunulması için `ServiceLoader` sınıfından faydalanacağız. Fakat bu metodların içerisini doldurmadan evvel örneğin bir varsayılan sağlayıcı nesnesi oluşturalım.

```
public class StandardRandomProvider implements `RandomServiceProvider` {

    @Override
    public Integer random(int start, int end) {
        Random random = new Random();
        int randomInt = random.nextInt((end - start + 1)) + start;
        return randomInt;
    }

}
```

Görüldüğü üzere `RandomServiceProvider` sağlayıcı arayüzünü uygulayan bir `StandardRandomProvider` sınıfı oluşturduk. Bu sınıf içerisindeki `random()` metodu, `Random` sınıfı üzerinden verilen aralıkta rastgele bir tamsayı üretmektedir.

Fakat servis sağlayıcı sınıfını yazmak yetmemektedir. Bunun önceden tanımlı bir biçimde mevcut uygulamaya tanıtılması gerekmektedir. Bunun için `StandardRandomProvider` sınıfının yer aldığı projede `META-INF/services` dizini altında `RandomServiceProvider` sınıfının tam adıyla bir dosya oluşturulmalı ve bu dosyanın içine `RandomServiceProvider` sınıfının tam adı yazılmalıdır.

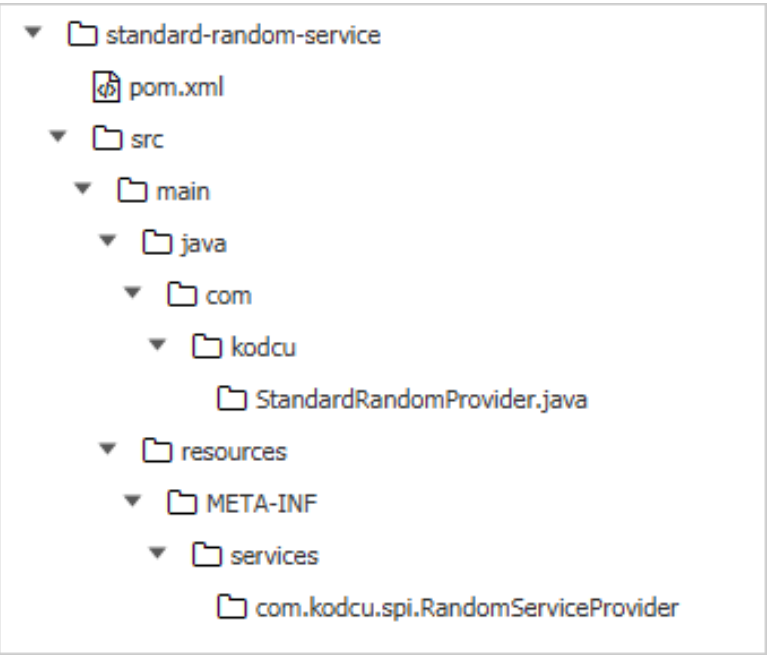


Figure 2. standard-random-service projesi

`com.kodcu.spi.RandomServiceProvider` dosyasını içi ise aşağıdaki gibidir.

com.kodcu.spi.RandomServiceProvider dosyasının içi

```
com.kodcu.StandardRandomProvider
```

Bu şekilde artık `StandardRandomProvider` servisini ServiceLoader sistemine göre yapılandırmış oluyoruz. Şimdi, `RandomServiceProvider` arayüzü içini uygulayıcılarını sunar şekilde yapılandırmaya devam edelim.

```
public interface RandomServiceProvider {

public Integer random(int start, int end);

public static RandomServiceProvider getDefaultProvider() {
    return getProvider("com.kodcu.StandardRandomProvider"); ❶
}

public static RandomServiceProvider getProvider(String providerName) {
    ServiceLoader<RandomServiceProvider> serviceLoader =
 ServiceLoader.load(RandomServiceProvider.class); ❷
```

```
    for (RandomServiceProvider provider : serviceLoader) { ❸
        String className = provider.getClass().getName(); ❹
        if (providerName.equals(className)) ❺
            return provider;
    }

    throw new RuntimeException(providerName + " provider is not found!"); ❻
}

}
```

❶ `com.kodcu.StandardRandomProvider` tam isimli varsayılan sağlayıcı sınıfını **CLASSPATH** 'den yükler ve döndürür.

❷ Yükleyeceği sağlayıcı arayüz türünden bir `ServiceLoader` nesnesi üretilir.

❸ **CLASSPATH** içinde bulunan tüm `RandomServiceProvider` servis sınıfları türünden nesneler turlanır.

❹ Bulunan `RandomServiceProvider` türünden sınıfın tam adı elde edilir.

❺ Eğer bulunan sınıfın tam adı metoda düşen isimle aynı ise ilgili nesne metotdan döndürülür.

❻ Eğer **CLASSPATH** içinde hiçbir servis uygulayıcısı bulunmadıysa bir istisna mesajı ile geliştirici bilgilendirilir.

Şimdi varsayılan `StandardRandomProvider` sağlayıcısına alternatif yeni bir sağlayıcı oluşturalım. İsmi `ThreadedRandomProvider` olsun.

```
public class ThreadedRandomProvider implements RandomServiceProvider {

    @Override
    public Integer random(int start, int end) {
        int randomInt = ThreadLocalRandom.current().nextInt(start,end);
        return randomInt;
    }
}
```

`ThreadedRandomProvider` sınıfı rastgele sayı üretme işini `Random` sınıfı yerine `ThreadLocalRandom` sınıfı üzerinden yapıyor. Tabiki bu servis sınıfını yine kendi projesinde `META-INF/services` altında yapılandırmalıyız.

Figure 3. threaded-random-service projesi

`com.kodcu.spi.RandomServiceProvider` dosyasını içi ise aşağıdaki gibidir.

com.kodcu.spi.RandomServiceProvider dosyasının içi

```
com.kodcu.ThreadedRandomProvider
```

Böylece `RandomServiceProvider` servisi `StandardRandomProvider` gibi `ThreadedRandomProvider` sınıfını da kullanıcılarına sunabilecektir.

Şimdi bu üç ayrı Java projeyesini `random-service-spi` , `standard-random-service` , `threaded-random-service` çalıştıran `random-service-app` adında bir proje oluşturalım. Bu proje `random-service-spi` servisi üzerinden iki servis sağlayıcısını yükleyecek ve `random()` metodlarını koşturacaktır.

```
public class App {

    public static void main(String[] args) {

        RandomServiceProvider defaultProvider =
 RandomServiceProvider.getDefaultProvider(); ❶
```

```
        System.out.println(defaultProvider.random(1,1000));  ❷

        RandomServiceProvider threadedProvider =
 RandomServiceProvider.getProvider("com.kodcu.ThreadedRandomProvider"); ❸

        System.out.println(threadedProvider.random(1,1000)); ❹
    }

}
```

❶ `RandomServiceProvider.defaultProvider()` metodu üzerinden varsayılan servis sağlayıcısını `com.kodcu.StandardRandomProvider` yükler.

❷ `StandardRandomProvider` 'ın `random()` metodu ile rastgele bir sayı üretip çıktılar.

❸ `RandomServiceProvider.getProvider()` metodu üzerinden `com.kodcu.ThreadedRandomProvider` servis sağlayıcısını yükler.

❹ `ThreadedRandomProvider` 'ın `random()` metodu ile rastgele bir sayı üretip çıktılar.

`random-service-app` projesinin dosya sistemindeki görünümü aşağıdaki gibidir.

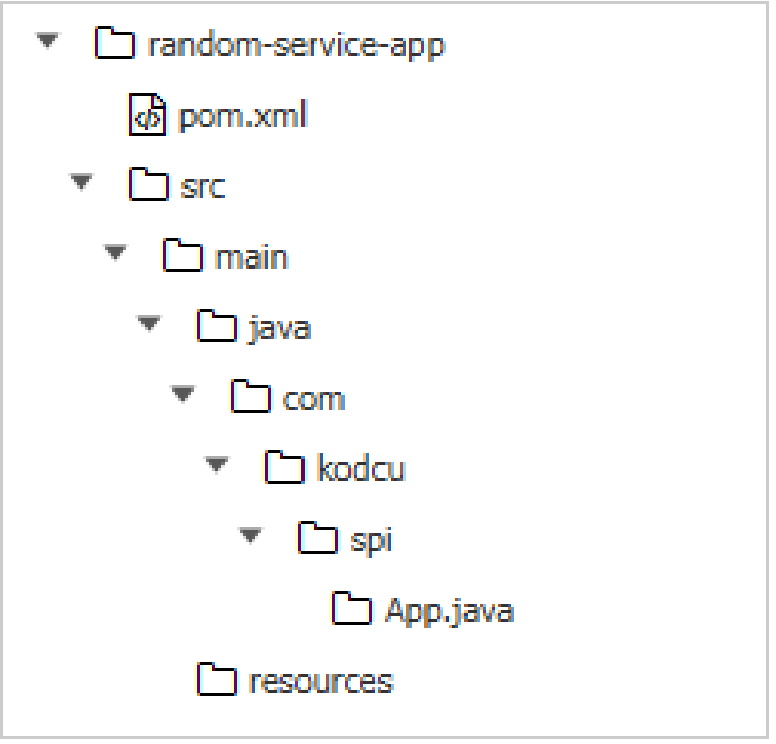
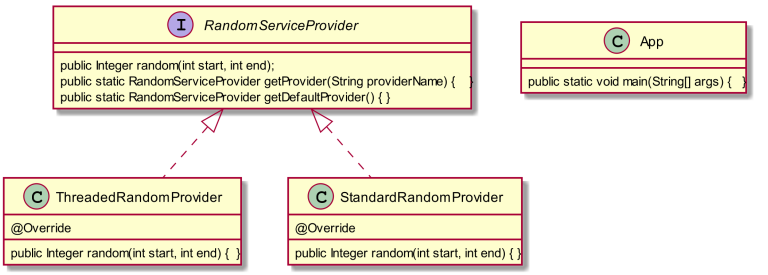


Figure 4. random-service-app projesi

Böylece standart bir servise, ekler halinde ayrı uygulayıcılar oluşturabilmekteyiz. `ServiceLoader` API ise biz geliştiricilere standart bir yol sunmuş oluyor.

Aşağıda servis ve iki uygulayıcısını ve bunları tüketen `App` sınıfını UML diagram halinde görebilirsiniz.



Kaynak kodlara buradaki[1] bağlantıdan erişebilirsiniz.

Tekrar görüşmek dileğiyle

[1] https://github.com/rahmanusta/java-service-spi-example