Bài 2

Resources

Built-in Resource Implementations

Spring includes several built-in `Resource` implementations:

- `UrlResource`
- `ClassPathResource`
- `FileSystemResource`
- `PathResource`
- `ServletContextResource`
- `InputStreamResource`
- `ByteArrayResource`

The ResourceLoader Interface

The `ResourceLoader` interface is meant to be implemented by objects that can return (that is, load) `Resource` instances. The following listing shows the `ResourceLoader` interface definition:

```java
public interface ResourceLoader {

    Resource getResource(String location);

    ClassLoader getClassLoader();
}
```

All application contexts implement the `ResourceLoader` interface. Therefore, all application contexts may be used to obtain `Resource` instances.

When you call `getResource()` on a specific application context, and the location path specified doesn't have a specific prefix, you get back a `Resource` type that is appropriate to that particular application context. For example, assume the following snippet of code was run against a `ClassPathXmlApplicationContext` instance:

Java Kotlin

```java
Resource template = ctx.getResource("some/resource/path/myTemplate.txt");
```

Against a `ClassPathXmlApplicationContext`, that code returns a `ClassPathResource`. If the same method were run against a `FileSystemXmlApplicationContext` instance, it would return a `FileSystemResource`. For a `WebApplicationContext`, it would return a `ServletContextResource`. It would similarly return appropriate objects for each context.

As a result, you can load resources in a fashion appropriate to the particular application context.

The ResourceLoader Interface

On the other hand, you may also force `ClassPathResource` to be used, regardless of the application context type, by specifying the special `classpath:` prefix, as the following example shows:

Java Kotlin

```java
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");
```

Similarly, you can force a `UrlResource` to be used by specifying any of the standard `java.net.URL` prefixes. The following examples use the `file` and `https` prefixes:

Java Kotlin

```java
Resource template = ctx.getResource("file:///some/resource/path/myTemplate.txt");
```

Java Kotlin

```java
Resource template = ctx.getResource("https://myhost.com/resource/path/myTemplate.txt");
```

The ResourceLoader Interface

Table: Resource strings

Prefix	Example	Explanation
classpath:	`classpath:com/myapp/config.xml`	Loaded from the classpath.
file:	`file:///data/config.xml`	Loaded as a `URL` from the filesystem. See also `FileSystemResource` Caveats.
https:	`https://myserver/logo.png`	Loaded as a `URL`.
(none)	`/data/config.xml`	Depends on the underlying `ApplicationContext`.

Application context and Resource path

An application context constructor (for a specific application context type) generally takes a string or array of strings as the location paths of the resources, such as XML files that make up the definition of the context.

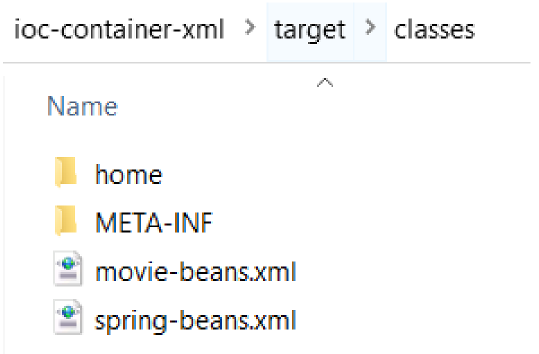
When such a location path does not have a prefix, the specific `Resource` type built from that path and used to load the bean definitions depends on and is appropriate to the specific application context. For example, consider the following example, which creates a `ClassPathXmlApplicationContext` :

Java Kotlin

JAVA
```java
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
```

Java Kotlin

JAVA
```java
ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:conf/appContext.xml"); as default
```


ioc-container-xml › target › classes

Name

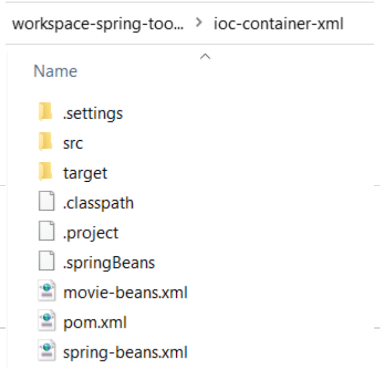
home
META-INF
movie-beans.xml
spring-beans.xml

.classpath

```xml
<classpathentry kind="src" output="target/classes" path="src/main/java">
    <attributes>
        <attribute name="optional" value="true"/>
        <attribute name="maven.pomderived" value="true"/>
    </attributes>
</classpathentry>
<classpathentry excluding="**" kind="src" output="target/classes" path="src/main/resources">
```

Application context and Resource path

The bean definitions are loaded from the classpath, because a `ClassPathResource` is used. However, consider the following example, which creates a `FileSystemXmlApplicationContext`:


workspace-spring-too... › ioc-container-xml

Name
.settings
src
target
.classpath
.project
.springBeans
movie-beans.xml
pom.xml
spring-beans.xml

Java | Kotlin

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/appContext.xml");
```

Now the bean definitions are loaded from a filesystem location (in this case, relative to the current working directory).

Note that the use of the special `classpath` prefix or a standard URL prefix on the location path overrides the default type of `Resource` created to load the bean definitions. Consider the following example:

Java | Kotlin

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

Using `FileSystemXmlApplicationContext` loads the bean definitions from the classpath. However, it is still a `FileSystemXmlApplicationContext`. If it is subsequently used as a `ResourceLoader`, any unprefixed paths are still treated as filesystem paths.

Application context and Resource path

Constructing `ClassPathXmlApplicationContext` Instances — Shortcuts

The `ClassPathXmlApplicationContext` exposes a number of constructors to enable convenient instantiation. The basic idea is that you can supply merely a string array that contains only the filenames of the XML files themselves (without the leading path information) and also supply a `Class`. The `ClassPathXmlApplicationContext` then derives the path information from the supplied class.

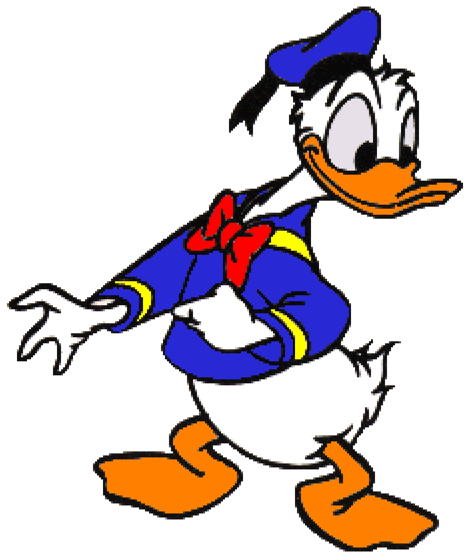
Consider the following directory layout:

```
com/
  example/
    services.xml
    repositories.xml
    MessengerService.class
```

The following example shows how a `ClassPathXmlApplicationContext` instance composed of the beans defined in files named `services.xml` and `repositories.xml` (which are on the classpath) can be instantiated:

Java Kotlin

```java
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    new String[] {"services.xml", "repositories.xml"}, MessengerService.class);
```



END