6. Resources

6.1 Introduction

Java's standard `java.net.URL` class and standard handlers for various URL prefixes unfortunately are not quite adequate enough for all access to low-level resources. For example, there is no standardized `URL` implementation that may be used to access a resource that needs to be obtained from the classpath, or relative to a `ServletContext`. While it is possible to register new handlers for specialized `URL` prefixes (similar to existing handlers for prefixes such as `http:`), this is generally quite complicated, and the `URL` interface still lacks some desirable functionality, such as a method to check for the existence of the resource being pointed to.

6.2 The `Resource` interface

Spring's `Resource` interface is meant to be a more capable interface for abstracting access to low-level resources.

```
public interface Resource extends InputStreamSource {

    boolean exists();

    boolean isOpen();

    URL getURL() throws IOException;

    File getFile() throws IOException;
```

```
    Resource createRelative(String relativePath) throws IOException;

    String getFilename();

    String getDescription();
}
```

```
public interface InputStreamSource {

    InputStream getInputStream() throws IOException;
}
```

Some of the most important methods from the `Resource` interface are:

- `getInputStream()`: locates and opens the resource, returning an `InputStream` for reading from the resource. It is expected that each invocation returns a fresh `InputStream`. It is the responsibility of the caller to close the stream.
- `exists()`: returns a `boolean` indicating whether this resource actually exists in physical form.
- `isOpen()`: returns a `boolean` indicating whether this resource represents a handle with an open stream. If `true`, the `InputStream` cannot be read multiple times, and must be read once only and then closed to avoid resource leaks. Will be `false` for all usual resource implementations, with the exception of `InputStreamResource`.
- `getDescription()`: returns a description for this resource, to be used for error output when working with the resource. This is often the fully qualified file name or the actual URL of the resource.

Other methods allow you to obtain an actual `URL` or `File` object representing the resource (if the underlying implementation is compatible, and supports that functionality).

The `Resource` abstraction is used extensively in Spring itself, as an argument type in many method signatures when a resource is needed. Other methods in some Spring APIs (such as the constructors to various `ApplicationContext` implementations), take a `String` which in unadorned or simple form is used to create a `Resource` appropriate to that context implementation, or via special prefixes on the `String` path, allow the caller to specify that a specific `Resource` implementation must be created and used.

While the `Resource` interface is used a lot with Spring and by Spring, it's actually very useful to use as a general utility class by itself in your own code, for access to resources, even when your code doesn't know or care about any other parts of Spring. While this couples your code to Spring, it really only couples it to this small set of utility classes, which are serving as a more capable replacement for `URL`, and can be considered equivalent to any other library you would use for this purpose.

It is important to note that the `Resource` abstraction does not replace functionality: it wraps it where possible. For example, a `UrlResource` wraps a URL, and uses the wrapped `URL` to do its work.

6.3 Built-in `Resource` implementations

There are a number of `Resource` implementations that come supplied straight out of the box in Spring:

6.3.1 `UrlResource`

The `UrlResource` wraps a `java.net.URL`, and may be used to access any object that is normally accessible via a URL, such as files, an HTTP target, an FTP target, etc. All URLs have a standardized `String` representation, such that appropriate standardized prefixes are used to indicate one URL type from another. This includes `file:` for accessing filesystem paths, `http:` for accessing resources via the HTTP protocol, `ftp:` for accessing resources via FTP, etc.

A `UrlResource` is created by Java code explicitly using the `UrlResource` constructor, but will often be created implicitly when you call an API method which takes a `String` argument which is meant to represent a path. For the latter case, a JavaBeans `PropertyEditor` will ultimately decide which type of `Resource` to create. If the path string contains a few well-known (to it, that is) prefixes such as `classpath:`, it will create an appropriate specialized `Resource` for that prefix. However, if it doesn't recognize the prefix, it will assume the this is just a standard URL string, and will create a `UrlResource`.

6.3.2 `ClassPathResource`

This class represents a resource which should be obtained from the classpath. This uses either the thread context class loader, a given class loader, or a given class for loading resources.

This `Resource` implementation supports resolution as `java.io.File` if the class path resource resides in the file system, but not for classpath resources which reside in a jar and have not been expanded (by the servlet engine, or whatever the environment is) to the filesystem. To address this the various `Resource` implementations always support resolution as a `java.net.URL`.

A `ClassPathResource` is created by Java code explicitly using the `ClassPathResource` constructor, but will often be created implicitly when you call an API method which takes a `String` argument which is meant to represent a path. For the latter case, a JavaBeans `PropertyEditor` will recognize the special prefix `classpath:` on the string path, and create a `ClassPathResource` in that case.

6.3.3 `FileSystemResource`

This is a `Resource` implementation for `java.io.File` handles. It obviously supports resolution as a `File`, and as a `URL`.

6.3.4 `ServletContextResource`

This is a `Resource` implementation for `ServletContext` resources, interpreting relative paths within the relevant web application's root directory.

This always supports stream access and URL access, but only allows `java.io.File` access when the web application archive is expanded and the resource is physically on the filesystem. Whether or not it's expanded and on the filesystem like this, or accessed directly from the JAR or somewhere else like a DB (it's conceivable) is actually dependent on the Servlet container.

6.3.5 `InputStreamResource`

A `Resource` implementation for a given `InputStream`. This should only be used if no specific `Resource` implementation is applicable. In particular, prefer `ByteArrayResource` or any of the file-based `Resource` implementations where possible.

In contrast to other `Resource` implementations, this is a descriptor for an *already* opened resource - therefore returning `true` from `isOpen()`. Do not use it if you need to keep the resource descriptor somewhere, or if you need to read a stream multiple

times.

6.3.6 `ByteArrayResource`

This is a `Resource` implementation for a given byte array. It creates a `ByteArrayInputStream` for the given byte array.

It's useful for loading content from any given byte array, without having to resort to a single-use `InputStreamResource`.

6.4 The `ResourceLoader`

The `ResourceLoader` interface is meant to be implemented by objects that can return (i.e. load) `Resource` instances.

```
public interface ResourceLoader {
    Resource getResource(String location);
}
```

All application contexts implement the `ResourceLoader` interface, and 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 will get back a `Resource` type that is appropriate to that particular application context. For example, assume the following snippet of code was executed against a `ClassPathXmlApplicationContext` instance:

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

What would be returned would be a `ClassPathResource`; if the same method was executed against a `FileSystemXmlApplicationContext` instance, you'd get back a `FileSystemResource`. For a `WebApplicationContext`, you'd get back a `ServletContextResource`, and so on.

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

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

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

Similarly, one can force a `UrlResource` to be used by specifying any of the standard `java.net.URL` prefixes:

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

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

The following table summarizes the strategy for converting `String`s to `Resource`s:

Table 6.1. 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. [1]
http:	`http://myserver/logo.png`	Loaded as a `URL`.
(none)	`/data/config.xml`	Depends on the underlying `ApplicationContext`.

[1] But see also Section 6.7.3, "`FileSystemResource` caveats".

6.5 The `ResourceLoaderAware` interface

The `ResourceLoaderAware` interface is a special marker interface, identifying objects that expect to be provided with a `ResourceLoader` reference.

```
public interface ResourceLoaderAware {

   void setResourceLoader(ResourceLoader resourceLoader);
}
```

When a class implements `ResourceLoaderAware` and is deployed into an application context (as a Spring-managed bean), it is recognized as `ResourceLoaderAware` by the application context. The application context will then invoke the `setResourceLoader(ResourceLoader)`, supplying itself as the argument (remember, all application contexts in Spring implement the `ResourceLoader` interface).

Of course, since an `ApplicationContext` is a `ResourceLoader`, the bean could also implement the `ApplicationContextAware` interface and use the supplied application context directly to load resources, but in general, it's better to use the specialized `ResourceLoader` interface if that's all that's needed. The code would just be coupled to the resource loading interface, which can be considered a utility interface, and not the whole Spring `ApplicationContext` interface.

As of Spring 2.5, you can rely upon autowiring of the `ResourceLoader` as an alternative to implementing the `ResourceLoaderAware` interface. The "traditional" `constructor` and `byType` autowiring modes (as described in Section 5.4.5, “Autowiring collaborators”) are now capable of providing a dependency of type `ResourceLoader` for either a constructor argument or setter method parameter respectively. For more flexibility (including the ability to autowire fields and multiple parameter methods), consider using the new annotation-based autowiring features. In that case, the `ResourceLoader` will be autowired into a field, constructor argument, or method parameter that is expecting the `ResourceLoader` type as long as the field, constructor, or method in question carries the `@Autowired` annotation. For more information, see Section 5.9.2, “`@Autowired`”.

6.6 `Resources` as dependencies

If the bean itself is going to determine and supply the resource path through some sort of dynamic process, it probably makes sense for the bean to use the `ResourceLoader` interface to load resources. Consider as an example the loading of a template of some sort, where the specific resource that is needed depends on the role of the user. If the resources are static, it makes sense to eliminate the use of the `ResourceLoader` interface completely, and just have the bean expose the `Resource` properties it needs, and expect that they will be injected into it.

What makes it trivial to then inject these properties, is that all application contexts register and use a special JavaBeans `PropertyEditor` which can convert `String` paths to `Resource` objects. So if `myBean` has a template property of type `Resource`, it can be configured with a simple string for that resource, as follows:

```
<bean id="myBean" class="...">
  <property name="template" value="some/resource/path/myTemplate.txt"/>
</bean>
```

Note that the resource path has no prefix, so because the application context itself is going to be used as the `ResourceLoader`, the resource itself will be loaded via a `ClassPathResource`, `FileSystemResource`, or `ServletContextResource` (as appropriate) depending on the exact type of the context.

If there is a need to force a specific `Resource` type to be used, then a prefix may be used. The following two examples show how to force a `ClassPathResource` and a `UrlResource` (the latter being used to access a filesystem file).

```
<property name="template" value="classpath:some/resource/path/myTemplate.txt">
```

```
<property name="template" value="file:/some/resource/path/myTemplate.txt"/>
```

6.7 Application contexts and `Resource` paths

6.7.1 Constructing application contexts

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

When such a location path doesn't 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, if you create a `ClassPathXmlApplicationContext` as follows:

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

The bean definitions will be loaded from the classpath, as a `ClassPathResource` will be used. But if you create a `FileSystemXmlApplicationContext` as follows:

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

The bean definition will be 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 will override the default type of `Resource` created to load the definition. So this `FileSystemXmlApplicationContext`...

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

... will actually load its bean definitions from the classpath. However, it is still a `FileSystemXmlApplicationContext`. If it is subsequently used as a `ResourceLoader`, any unprefixed paths will still be treated as filesystem paths.

Constructing `ClassPathXmlApplicationContext` instances - shortcuts

The `ClassPathXmlApplicationContext` exposes a number of constructors to enable convenient instantiation. The basic idea is that one supplies merely a string array containing just the filenames of the XML files themselves (without the leading path information), and one *also* supplies a `Class`; the `ClassPathXmlApplicationContext` will derive the path information from the supplied class.

An example will hopefully make this clear. Consider a directory layout that looks like this:

```
com/
  foo/
    services.xml
    daos.xml
    MessengerService.class
```

A `ClassPathXmlApplicationContext` instance composed of the beans defined in the `'services.xml'` and `'daos.xml'` could be instantiated like so...

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

Please do consult the Javadocs for the `ClassPathXmlApplicationContext` class for details of the various constructors.

6.7.2 Wildcards in application context constructor resource paths

The resource paths in application context constructor values may be a simple path (as shown above) which has a one-to-one mapping to a target Resource, or alternately may contain the special "classpath*:" prefix and/or internal Ant-style regular expressions (matched using Spring's `PathMatcher` utility). Both of the latter are effectively wildcards

One use for this mechanism is when doing component-style application assembly. All components can 'publish' context definition fragments to a well-known location path, and when the final application context is created using the same path prefixed via `classpath*:`, all component fragments will be picked up automatically.

Note that this wildcarding is specific to use of resource paths in application context constructors (or when using the `PathMatcher` utility class hierarchy directly), and is resolved at construction time. It has nothing to do with the `Resource` type itself. It's not possible to use the `classpath*:` prefix to construct an actual `Resource`, as a resource points to just one resource at a time.

Ant-style Patterns

When the path location contains an Ant-style pattern, for example:

```
/WEB-INF/*-context.xml
com/mycompany/**/applicationContext.xml
file:C:/some/path/*-context.xml
classpath:com/mycompany/**/applicationContext.xml
```

... the resolver follows a more complex but defined procedure to try to resolve the wildcard. It produces a Resource for the path up to the last non-wildcard segment and obtains a URL from it. If this URL is not a "jar:" URL or container-specific variant (e.g. "`zip:`" in WebLogic, "`wsjar`" in WebSphere, etc.), then a `java.io.File` is obtained from it and used to resolve the wildcard by traversing the filesystem. In the case of a jar URL, the resolver either gets a `java.net.JarURLConnection` from it or manually parses the jar URL and then traverses the contents of the jar file to resolve the wildcards.

Implications on portability

If the specified path is already a file URL (either explicitly, or implicitly because the base `ResourceLoader` is a filesystem one, then wildcarding is guaranteed to work in a completely portable fashion.

If the specified path is a classpath location, then the resolver must obtain the last non-wildcard path segment URL via a `Classloader.getResource()` call. Since this is just a node of the path (not the file at the end) it is actually undefined (in the `ClassLoader` Javadocs) exactly what sort of a URL is returned in this case. In practice, it is always a `java.io.File` representing the directory, where the classpath resource resolves to a filesystem location, or a jar URL of some sort, where the classpath resource resolves to a jar location. Still, there is a portability concern on this operation.

If a jar URL is obtained for the last non-wildcard segment, the resolver must be able to get a `java.net.JarURLConnection` from it, or manually parse the jar URL, to be able to walk the contents of the jar, and resolve the wildcard. This will work in most environments, but will fail in others, and it is strongly recommended that the wildcard resolution of resources coming from jars be thoroughly tested in your specific environment before you rely on it.

The `classpath*:` prefix

When constructing an XML-based application context, a location string may use the special `classpath*:` prefix:

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

This special prefix specifies that all classpath resources that match the given name must be obtained (internally, this essentially happens via a `ClassLoader.getResources(...)` call), and then merged to form the final application context definition.

The wildcard classpath relies on the `getResources()` method of the underlying classloader. As most application servers nowadays supply their own classloader implementation, the behavior might differ especially when dealing with jar files. A simple test to check if `classpath*` works is to use the classloader to load a file from within a jar on the classpath: `getClass().getClassLoader().getResources("<someFileInsideTheJar>")`. Try this test with files that have the same name but are placed inside two different locations. In case an inappropriate result is returned, check the application server documentation for settings that might affect the classloader behavior.

The "`classpath*:`" prefix can also be combined with a `PathMatcher` pattern in the rest of the location path, for example "`classpath*:META-INF/*-beans.xml`". In this case, the resolution strategy is fairly simple: a ClassLoader.getResources() call is used on the last non-wildcard path segment to get all the matching resources in the class loader hierarchy, and then off each resource the same PathMatcher resolution strategy described above is used for the wildcard subpath.

Other notes relating to wildcards

Please note that "`classpath*:`" when combined with Ant-style patterns will only work reliably with at least one root directory before the pattern starts, unless the actual target files reside in the file system. This means that a pattern like "`classpath*:*.xml`" will not retrieve files from the root of jar files but rather only from the root of expanded directories. This originates from a limitation in the JDK's `ClassLoader.getResources()` method which only returns file system locations for a passed-in empty string (indicating potential roots to search).

Ant-style patterns with "`classpath:`" resources are not guaranteed to find matching resources if the root package to search is available in multiple class path locations. This is because a resource such as

```
    com/mycompany/package1/service-context.xml
```

may be in only one location, but when a path such as

```
    classpath:com/mycompany/**/service-context.xml
```

is used to try to resolve it, the resolver will work off the (first) URL returned by `getResource("com/mycompany")`;. If this base package node exists in multiple classloader locations, the actual end resource may not be underneath. Therefore, preferably, use "`classpath*:`" with the same Ant-style pattern in such a case, which will search all class path locations that contain the root package.

6.7.3 `FileSystemResource` caveats

A `FileSystemResource` that is not attached to a `FileSystemApplicationContext` (that is, a `FileSystemApplicationContext` is not the actual `ResourceLoader`) will treat absolute vs. relative paths as you would expect. Relative paths are relative to the current working directory, while absolute paths are relative to the root of the filesystem.

For backwards compatibility (historical) reasons however, this changes when the `FileSystemApplicationContext` is the `ResourceLoader`. The `FileSystemApplicationContext` simply forces all attached `FileSystemResource` instances to treat all location paths as relative, whether they start with a leading slash or not. In practice, this means the following are equivalent:

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

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

As are the following: (Even though it would make sense for them to be different, as one case is relative and the other absolute.)

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("some/resource/path/myTemplate.txt");
```

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("/some/resource/path/myTemplate.txt");
```

In practice, if true absolute filesystem paths are needed, it is better to forgo the use of absolute paths with `FileSystemResource` / `FileSystemXmlApplicationContext`, and just force the use of a `UrlResource`, by using the `file:` URL prefix.

```
// actual context type doesn't matter, the Resource will always be UrlResource
ctx.getResource("file:/some/resource/path/myTemplate.txt");
```

```
// force this FileSystemXmlApplicationContext to load its definition via a UrlResource
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("file:/conf/context.xml");
```