Container Overview

The `org.springframework.context.ApplicationContext` interface represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on the components to instantiate, configure, and assemble by reading

Edit this Page
GitHub Project
Stack Overflow

components.

Several implementations of the `ApplicationContext` interface are part of core Spring. In stand-alone applications, it is common to create an instance of `AnnotationConfigApplicationContext` or `ClassPathXmlApplicationContext`.

In most application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a plain web application scenario, a simple boilerplate web descriptor XML in the `web.xml` file of the application suffices (see Convenient ApplicationContext Instantiation for Web Applications). In a Spring Boot scenario, the application context is implicitly bootstrapped for you based on common setup conventions.

The following diagram shows a high-level view of how Spring works. Your application classes are combined with configuration metadata so that, after the `ApplicationContext` is created and initialized, you have a fully configured and executable system or application.

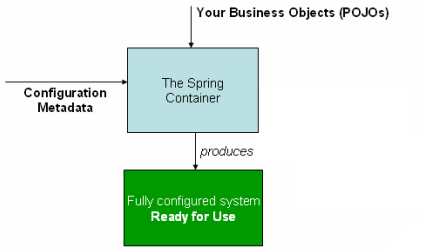

Figure 1. The Spring IoC container

Configuration Metadata

As the preceding diagram shows, the Spring IoC container consumes a form of configuration metadata. This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the components in your application.

The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. These days, many developers choose Java-based configuration for their Spring applications:

- Annotation-based configuration: define beans using annotation-based configuration metadata on your application's component classes.
- Java-based configuration: define beans external to your application classes by using Java-based configuration classes. To use these features, see the `@Configuration`, `@Bean`, `@Import`, and `@DependsOn` annotations.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage. Java configuration typically uses `@Bean`-annotated methods within a `@Configuration` class, each corresponding to one bean definition.

These bean definitions correspond to the actual objects that make up your application. Typically, you define service layer objects, persistence layer objects such as repositories or data access objects (DAOs), presentation objects such as Web controllers, infrastructure objects such as a JPA `EntityManagerFactory`, JMS queues, and so forth. Typically, one does not configure fine-grained domain objects in the container, because it is usually the responsibility of repositories and business logic to create and load domain objects.

XML as an External Configuration DSL

XML-based configuration metadata configures these beans as `<bean/>` elements inside a top-level `<beans/>` element. The following example shows the basic structure of XML-based configuration metadata:

```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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="..." class="..."> (1) (2)
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions go here -->

</beans>
```

(1) The `id` attribute is a string that identifies the individual bean definition.

(2) The `class` attribute defines the type of the bean and uses the fully qualified class name.

The value of the `id` attribute can be used to refer to collaborating objects. The XML for referring to collaborating objects is not shown in this example. See Dependencies for more information.

For instantiating a container, the location path or paths to the XML resource files need to be supplied to a `ClassPathXmlApplicationContext` constructor that let the container load configuration metadata from a variety of external resources, such as the local file system, the Java `CLASSPATH`, and so on.

Java Kotlin

```java
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml", "daos.xml");
```

NOTE

After you learn about Spring's IoC container, you may want to know more about Spring's `Resource` abstraction (as described in Resources) which

After you learn about Spring's IoC container, you may want to know more about Spring's `Resource` abstraction (as described in Resources) which provides a convenient mechanism for reading an InputStream from locations defined in a URI syntax. In particular, `Resource` paths are used to construct applications contexts, as described in Application Contexts and Resource Paths.

The following example shows the service layer objects `(services.xml)` configuration file:

```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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- services -->

    <bean id="petStore" class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions for services go here -->

</beans>
```

The following example shows the data access objects `daos.xml` file:

```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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="accountDao"
        class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <bean id="itemDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions for data access objects go here -->

</beans>
```

In the preceding example, the service layer consists of the `PetStoreServiceImpl` class and two data access objects of the types `JpaAccountDao` and `JpaItemDao` (based on the JPA Object-Relational Mapping standard). The `property name` element refers to the name of the JavaBean property, and the `ref` element refers to the name of another bean definition. This linkage between `id` and `ref` elements expresses the dependency between collaborating objects. For details of configuring an object's dependencies, see Dependencies.

Composing XML-based Configuration Metadata

It can be useful to have bean definitions span multiple XML files. Often, each individual XML configuration file represents a logical layer or module in your architecture.

You can use the `ClassPathXmlApplicationContext` constructor to load bean definitions from XML fragments. This constructor takes multiple `Resource` locations, as was shown in the previous section. Alternatively, use one or more occurrences of the `<import/>` element to load bean definitions from another file or files. The following example shows how to do so:

```xml
<beans>
    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
</beans>
```

In the preceding example, external bean definitions are loaded from three files: `services.xml`, `messageSource.xml`, and `themeSource.xml`. All location paths are relative to the definition file doing the importing, so `services.xml` must be in the same directory or classpath location as the file doing the importing, while `messageSource.xml` and `themeSource.xml` must be in a `resources` location below the location of the importing file. As you can see, a leading slash is ignored. However, given that these paths are relative, it is better form not to use the slash at all. The contents of the files being imported, including the top level `<beans/>` element, must be valid XML bean definitions, according to the Spring Schema.

NOTE

It is possible, but not recommended, to reference files in parent directories using a relative "../" path. Doing so creates a dependency on a file that is outside the current application. In particular, this reference is not recommended for `classpath:` URLs (for example, `classpath:../services.xml`), where the runtime resolution process chooses the "nearest" classpath root and then looks into its parent directory. Classpath configuration changes may lead to the choice of a different, incorrect directory.

You can always use fully qualified resource locations instead of relative paths: for example, `file:C:/config/services.xml` or `classpath:/config/services.xml`. However, be aware that you are coupling your application's configuration to specific absolute locations. It is generally preferable to keep an indirection for such absolute locations — for example, through "${…}" placeholders that are resolved against JVM system properties at runtime.

The namespace itself provides the import directive feature. Further configuration features beyond plain bean definitions are available in a selection of XML namespaces provided by Spring — for example, the `context` and `util` namespaces.

The Groovy Bean Definition DSL

As a further example for externalized configuration metadata, bean definitions can also be expressed in Spring's Groovy Bean Definition DSL, as known from the Grails framework. Typically, such configuration live in a ".groovy" file with the structure shown in the following example:

```groovy
beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
        settings = [mynew:"setting"]
```

```
    }
    sessionFactory(SessionFactory) {
        dataSource = dataSource
    }
    myService(MyService) {
        nestedBean = { AnotherBean bean ->
            dataSource = dataSource
        }
    }
}
```

This configuration style is largely equivalent to XML bean definitions and even supports Spring's XML configuration namespaces. It also allows for importing XML bean definition files through an `importBeans` directive.

Using the Container

The `ApplicationContext` is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. By using the method `T getBean(String name, Class<T> requiredType)`, you can retrieve instances of your beans.

The `ApplicationContext` lets you read bean definitions and access them, as the following example shows:

Java Kotlin

```
// create and configure beans
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml", "daos.xml");

// retrieve configured instance
PetStoreService service = context.getBean("petStore", PetStoreService.class);

// use configured instance
List<String> userList = service.getUsernameList();
```

With Groovy configuration, bootstrapping looks very similar. It has a different context implementation class which is Groovy-aware (but also understands XML bean definitions). The following example shows Groovy configuration:

Java Kotlin

```
ApplicationContext context = new GenericGroovyApplicationContext("services.groovy", "daos.groovy");
```

The most flexible variant is `GenericApplicationContext` in combination with reader delegates — for example, with `XmlBeanDefinitionReader` for XML files, as the following example shows:

Java Kotlin

```
GenericApplicationContext context = new GenericApplicationContext();
new XmlBeanDefinitionReader(context).loadBeanDefinitions("services.xml", "daos.xml");
context.refresh();
```

You can also use the `GroovyBeanDefinitionReader` for Groovy files, as the following example shows:

Java Kotlin

```
GenericApplicationContext context = new GenericApplicationContext();
new GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy", "daos.groovy");
context.refresh();
```

You can mix and match such reader delegates on the same `ApplicationContext`, reading bean definitions from diverse configuration sources.

You can then use `getBean` to retrieve instances of your beans. The `ApplicationContext` interface has a few other methods for retrieving beans, but, ideally, your application code should never use them. Indeed, your application code should have no calls to the `getBean()` method at all and thus have no dependency on Spring APIs at all. For example, Spring's integration with web frameworks provides dependency injection for various web framework components such as controllers and JSF-managed beans, letting you declare a dependency on a specific bean through metadata (such as an autowiring annotation).