# Spring Framework Documentation

Version 5.0.8.RELEASE

Welcome to the Spring Framework reference documentation!

Please read the Overview for a quick introduction including brief history, design philosophy, where to ask questions, and tips to get started. For information on "What’s New", or "Migrating from previous versions", check the Github Wiki.

The reference documentation is divided into several sections:

Core

IoC container, Events, Resources, i18n, Validation, Data Binding, Type Conversion, SpEL, AOP.

Testing

Mock objects, TestContext framework, Spring MVC Test, WebTestClient.

Data Access

Transactions, DAO support, JDBC, ORM, Marshalling XML.

Web Servlet

Spring MVC, WebSocket, SockJS, STOMP messaging.

Web Reactive

Spring WebFlux, WebClient, WebSocket.

Integration

Remoting, JMS, JCA, JMX, Email, Tasks, Scheduling, Cache.

Languages

Kotlin, Groovy, Dynamic languages.

## Spring Framework Overview

Version 5.0.8.RELEASE

Spring makes it easy to create Java enterprise applications. It provides everything you need to embrace the Java language in an enterprise environment, with support for Groovy and Kotlin as alternative languages on the JVM, and with the flexibility to create many kinds of architectures depending on an application’s needs. As of Spring Framework 5.0, Spring requires JDK 8+ (Java SE 8+) and provides out-of-the-box support for JDK 9 already.

Spring supports a wide range of application scenarios. In a large enterprise, applications often exist for a long time and have to run on a JDK and application server whose upgrade cycle is beyond developer control. Others may run as a single jar with the server embedded, possibly in a cloud environment. Yet others may be standalone applications (such as batch or integration workloads) that do not need a server.

Spring is open source. It has a large and active community that provides continuous feedback based on a diverse range of real-world use cases. This has helped Spring to successfully evolve over a very long time.

1. What We Mean by "Spring"

The term "Spring" means different things in different contexts. It can be used to refer to the Spring Framework project itself, which is where it all started. Over time, other Spring projects have been built on top of the Spring Framework. Most often, when people say "Spring", they mean the entire family of projects. This reference documentation focuses on the foundation: the Spring Framework itself.

The Spring Framework is divided into modules. Applications can choose which modules they need. At the heart are the modules of the core container, including a configuration model and a dependency injection mechanism. Beyond that, the Spring Framework provides foundational support for different application architectures, including messaging, transactional data and persistence, and web. It also includes the Servlet-based Spring MVC web framework and, in parallel, the Spring WebFlux reactive web framework.

A note about modules: Spring’s framework jars allow for deployment to JDK 9’s module path ("Jigsaw"). For use in Jigsaw-enabled applications, the Spring Framework 5 jars come with "Automatic-Module-Name" manifest entries which define stable language-level module names ("spring.core", "spring.context" etc) independent from jar artifact names (the jars follow the same naming pattern with "-" instead of ".", e.g. "spring-core" and "spring-context"). Of course, Spring’s framework jars keep working fine on the classpath on both JDK 8 and 9.

2. History of Spring and the Spring Framework

Spring came into being in 2003 as a response to the complexity of the early J2EE specifications. While some consider Java EE and Spring to be in competition, Spring is, in fact, complementary to Java EE. The Spring programming model does not embrace the Java EE platform specification; rather, it integrates with carefully selected individual specifications from the EE umbrella:

Servlet API (JSR 340)

WebSocket API (JSR 356)

Concurrency Utilities (JSR 236)

JSON Binding API (JSR 367)

Bean Validation (JSR 303)

JPA (JSR 338)

JMS (JSR 914)

as well as JTA/JCA setups for transaction coordination, if necessary.

The Spring Framework also supports the Dependency Injection (JSR 330) and Common Annotations (JSR 250) specifications, which application developers may choose to use instead of the Spring-specific mechanisms provided by the Spring Framework.

As of Spring Framework 5.0, Spring requires the Java EE 7 level (e.g. Servlet 3.1+, JPA 2.1+) as a minimum - while at the same time providing out-of-the-box integration with newer APIs at the Java EE 8 level (e.g. Servlet 4.0, JSON Binding API) when encountered at runtime. This keeps Spring fully compatible with e.g. Tomcat 8 and 9, WebSphere 9, and JBoss EAP 7.

Over time, the role of Java EE in application development has evolved. In the early days of Java EE and Spring, applications were created to be deployed to an application server. Today, with the help of Spring Boot, applications are created in a devops- and cloud-friendly way, with the Servlet container embedded and trivial to change. As of Spring Framework 5, a WebFlux application does not even use the Servlet API directly and can run on servers (such as Netty) that are not Servlet containers.

Spring continues to innovate and to evolve. Beyond the Spring Framework, there are other projects, such as Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch, among others. It’s important to remember that each project has its own source code repository, issue tracker, and release cadence. See spring.io/projects for the complete list of Spring projects.

3. Design Philosophy

When you learn about a framework, it’s important to know not only what it does but what principles it follows. Here are the guiding principles of the Spring Framework:

Provide choice at every level. Spring lets you defer design decisions as late as possible. For example, you can switch persistence providers through configuration without changing your code. The same is true for many other infrastructure concerns and integration with third-party APIs.

Accommodate diverse perspectives. Spring embraces flexibility and is not opinionated about how things should be done. It supports a wide range of application needs with different perspectives.

Maintain strong backward compatibility. Spring’s evolution has been carefully managed to force few breaking changes between versions. Spring supports a carefully chosen range of JDK versions and third-party libraries to facilitate maintenance of applications and libraries that depend on Spring.

Care about API design. The Spring team puts a lot of thought and time into making APIs that are intuitive and that hold up across many versions and many years.

Set high standards for code quality. The Spring Framework puts a strong emphasis on meaningful, current, and accurate Javadoc. It is one of very few projects that can claim clean code structure with no circular dependencies between packages.

4. Feedback and Contributions

For how-to questions or diagnosing or debugging issues, we suggest using StackOverflow, and we have a questions page that lists the suggested tags to use. If you’re fairly certain that there is a problem in the Spring Framework or would like to suggest a feature, please use the JIRA issue tracker.

If you have a solution in mind or a suggested fix, you can submit a pull request on Github. However, please keep in mind that, for all but the most trivial issues, we expect a ticket to be filed in the issue tracker, where discussions take place and leave a record for future reference.

For more details see the guidelines at the CONTRIBUTING, top-level project page.

5. Getting Started

If you are just getting started with Spring, you may want to begin using the Spring Framework by creating a Spring Boot-based application. Spring Boot provides a quick (and opinionated) way to create a production-ready Spring-based application. It is based on the Spring Framework, favors convention over configuration, and is designed to get you up and running as quickly as possible.

You can use start.spring.io to generate a basic project or follow one of the "Getting Started" guides, such as Getting Started Building a RESTful Web Service. As well as being easier to digest, these guides are very task focused, and most of them are based on Spring Boot. They also cover other projects from the Spring portfolio that you might want to consider when solving a particular problem.

## Core Technologies

Version 5.0.8.RELEASE

This part of the reference documentation covers all of those technologies that are absolutely integral to the Spring Framework.

Foremost amongst these is the Spring Framework’s Inversion of Control (IoC) container. A thorough treatment of the Spring Framework’s IoC container is closely followed by comprehensive coverage of Spring’s Aspect-Oriented Programming (AOP) technologies. The Spring Framework has its own AOP framework, which is conceptually easy to understand, and which successfully addresses the 80% sweet spot of AOP requirements in Java enterprise programming.

Coverage of Spring’s integration with AspectJ (currently the richest - in terms of features - and certainly most mature AOP implementation in the Java enterprise space) is also provided.

1. The IoC container

1.1. Introduction to the Spring IoC container and beans

This chapter covers the Spring Framework implementation of the Inversion of Control (IoC) [1] principle. IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the Service Locator pattern.

The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework’s IoC container. The BeanFactory interface provides an advanced configuration mechanism capable of managing any type of object. ApplicationContext is a sub-interface of BeanFactory. It adds easier integration with Spring’s AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the WebApplicationContext for use in web applications.

In short, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality. The ApplicationContext is a complete superset of the BeanFactory, and is used exclusively in this chapter in descriptions of Spring’s IoC container. For more information on using the BeanFactory instead of the ApplicationContext, refer to The BeanFactory.

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

1.2. Container overview

The interface org.springframework.context.ApplicationContext represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the aforementioned beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It allows you to express the objects that compose your application and the rich interdependencies between such objects.

Several implementations of the ApplicationContext interface are supplied out-of-the-box with Spring. In standalone applications it is common to create an instance of ClassPathXmlApplicationContext or FileSystemXmlApplicationContext. While XML has been the traditional format for defining configuration metadata you can instruct the container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

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 web application scenario, a simple eight (or so) lines of boilerplate web descriptor XML in the web.xml file of the application will typically suffice (see Convenient ApplicationContext instantiation for web applications). If you are using the Spring Tool Suite Eclipse-powered development environment this boilerplate configuration can be easily created with few mouse clicks or keystrokes.

The following diagram is 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.

container magic

Figure 1. The Spring IoC container

1.2.1. 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 objects in your application.

Configuration metadata is traditionally supplied in a simple and intuitive XML format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container.

XML-based metadata is not the only allowed form of configuration metadata. 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.

For information about using other forms of metadata with the Spring container, see:

Annotation-based configuration: Spring 2.5 introduced support for annotation-based configuration metadata.

Java-based configuration: Starting with Spring 3.0, many features provided by the Spring JavaConfig project became part of the core Spring Framework. Thus you can define beans external to your application classes by using Java rather than XML files. To use these new 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. XML-based configuration metadata shows these beans configured as <bean/> elements inside a top-level <beans/> element. Java configuration typically uses @Bean annotated methods within a @Configuration class.

These bean definitions correspond to the actual objects that make up your application. Typically you define service layer objects, data access objects (DAOs), presentation objects such as Struts Action instances, infrastructure objects such as Hibernate SessionFactories, JMS Queues, and so forth. Typically one does not configure fine-grained domain objects in the container, because it is usually the responsibility of DAOs and business logic to create and load domain objects. However, you can use Spring’s integration with AspectJ to configure objects that have been created outside the control of an IoC container. See Using AspectJ to dependency-inject domain objects with Spring.

The following example shows the basic structure of XML-based configuration metadata:

<?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="..." class="...">

<!-- 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>

The id attribute is a string that you use to identify the individual bean definition. The class attribute defines the type of the bean and uses the fully qualified classname. The value of the id attribute refers to collaborating objects. The XML for referring to collaborating objects is not shown in this example; see Dependencies for more information.

1.2.2. Instantiating a container

Instantiating a Spring IoC container is straightforward. The location path or paths supplied to an ApplicationContext constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java CLASSPATH, and so on.

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

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 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">

<!-- 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 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="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 class PetStoreServiceImpl, and two data access objects of the type 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 application context constructor to load bean definitions from all these 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. For example:

<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, but 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.

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 import directive is a feature provided by the beans namespace itself. Further configuration features beyond plain bean definitions are available in a selection of XML namespaces provided by Spring, e.g. the "context" and the "util" namespace.

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 will live in a ".groovy" file with a structure as follows:

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.

1.2.3. Using the container

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

The ApplicationContext enables you to read bean definitions and access them as follows:

// 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, just a different context implementation class which is Groovy-aware (but also understands XML bean definitions):

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

The most flexible variant is GenericApplicationContext in combination with reader delegates, e.g. with XmlBeanDefinitionReader for XML files:

GenericApplicationContext context = new GenericApplicationContext();

new XmlBeanDefinitionReader(context).loadBeanDefinitions("services.xml", "daos.xml");

context.refresh();

Or with GroovyBeanDefinitionReader for Groovy files:

GenericApplicationContext context = new GenericApplicationContext();

new GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy", "daos.groovy");

context.refresh();

Such reader delegates can be mixed and matched on the same ApplicationContext, reading bean definitions from diverse configuration sources, if desired.

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 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, allowing you to declare a dependency on a specific bean through metadata (e.g. an autowiring annotation).

1.3. Bean overview

A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container, for example, in the form of XML <bean/> definitions.

Within the container itself, these bean definitions are represented as BeanDefinition objects, which contain (among other information) the following metadata:

A package-qualified class name: typically the actual implementation class of the bean being defined.

Bean behavioral configuration elements, which state how the bean should behave in the container (scope, lifecycle callbacks, and so forth).

References to other beans that are needed for the bean to do its work; these references are also called collaborators or dependencies.

Other configuration settings to set in the newly created object, for example, the number of connections to use in a bean that manages a connection pool, or the size limit of the pool.

This metadata translates to a set of properties that make up each bean definition.

Table 1. The bean definition

Property Explained in…​

class

Instantiating beans

name

Naming beans

scope

Bean scopes

constructor arguments

Dependency Injection

properties

Dependency Injection

autowiring mode

Autowiring collaborators

lazy-initialization mode

Lazy-initialized beans

initialization method

Initialization callbacks

destruction method

Destruction callbacks

In addition to bean definitions that contain information on how to create a specific bean, the ApplicationContext implementations also permit the registration of existing objects that are created outside the container, by users. This is done by accessing the ApplicationContext’s BeanFactory via the method getBeanFactory() which returns the BeanFactory implementation DefaultListableBeanFactory. DefaultListableBeanFactory supports this registration through the methods registerSingleton(..) and registerBeanDefinition(..). However, typical applications work solely with beans defined through metadata bean definitions.

Bean metadata and manually supplied singleton instances need to be registered as early as possible, in order for the container to properly reason about them during autowiring and other introspection steps. While overriding of existing metadata and existing singleton instances is supported to some degree, the registration of new beans at runtime (concurrently with live access to factory) is not officially supported and may lead to concurrent access exceptions and/or inconsistent state in the bean container.

1.3.1. Naming beans

Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean. A bean usually has only one identifier, but if it requires more than one, the extra ones can be considered aliases.

In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s). The id attribute allows you to specify exactly one id. Conventionally these names are alphanumeric ('myBean', 'fooService', etc.), but may contain special characters as well. If you want to introduce other aliases to the bean, you can also specify them in the name attribute, separated by a comma (,), semicolon (;), or white space. As a historical note, in versions prior to Spring 3.1, the id attribute was defined as an xsd:ID type, which constrained possible characters. As of 3.1, it is defined as an xsd:string type. Note that bean id uniqueness is still enforced by the container, though no longer by XML parsers.

You are not required to supply a name or id for a bean. If no name or id is supplied explicitly, the container generates a unique name for that bean. However, if you want to refer to that bean by name, through the use of the ref element or Service Locator style lookup, you must provide a name. Motivations for not supplying a name are related to using inner beans and autowiring collaborators.

Bean Naming Conventions

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter, and are camel-cased from then on. Examples of such names would be (without quotes) 'accountManager', 'accountService', 'userDao', 'loginController', and so forth.

Naming beans consistently makes your configuration easier to read and understand, and if you are using Spring AOP it helps a lot when applying advice to a set of beans related by name.

With component scanning in the classpath, Spring generates bean names for unnamed components, following the rules above: essentially, taking the simple class name and turning its initial character to lower-case. However, in the (unusual) special case when there is more than one character and both the first and second characters are upper case, the original casing gets preserved. These are the same rules as defined by java.beans.Introspector.decapitalize (which Spring is using here).

Aliasing a bean outside the bean definition

In a bean definition itself, you can supply more than one name for the bean, by using a combination of up to one name specified by the id attribute, and any number of other names in the name attribute. These names can be equivalent aliases to the same bean, and are useful for some situations, such as allowing each component in an application to refer to a common dependency by using a bean name that is specific to that component itself.

Specifying all aliases where the bean is actually defined is not always adequate, however. It is sometimes desirable to introduce an alias for a bean that is defined elsewhere. This is commonly the case in large systems where configuration is split amongst each subsystem, each subsystem having its own set of object definitions. In XML-based configuration metadata, you can use the <alias/> element to accomplish this.

<alias name="fromName" alias="toName"/>

In this case, a bean in the same container which is named fromName, may also, after the use of this alias definition, be referred to as toName.

For example, the configuration metadata for subsystem A may refer to a DataSource via the name subsystemA-dataSource. The configuration metadata for subsystem B may refer to a DataSource via the name subsystemB-dataSource. When composing the main application that uses both these subsystems the main application refers to the DataSource via the name myApp-dataSource. To have all three names refer to the same object you add to the MyApp configuration metadata the following aliases definitions:

<alias name="subsystemA-dataSource" alias="subsystemB-dataSource"/>

<alias name="subsystemA-dataSource" alias="myApp-dataSource" />

Now each component and the main application can refer to the dataSource through a name that is unique and guaranteed not to clash with any other definition (effectively creating a namespace), yet they refer to the same bean.

Java-configuration

If you are using Java-configuration, the @Bean annotation can be used to provide aliases see Using the @Bean annotation for details.

1.3.2. Instantiating beans

A bean definition essentially is a recipe for creating one or more objects. The container looks at the recipe for a named bean when asked, and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object.

If you use XML-based configuration metadata, you specify the type (or class) of object that is to be instantiated in the class attribute of the <bean/> element. This class attribute, which internally is a Class property on a BeanDefinition instance, is usually mandatory. (For exceptions, see Instantiation using an instance factory method and Bean definition inheritance.) You use the Class property in one of two ways:

Typically, to specify the bean class to be constructed in the case where the container itself directly creates the bean by calling its constructor reflectively, somewhat equivalent to Java code using the new operator.

To specify the actual class containing the static factory method that will be invoked to create the object, in the less common case where the container invokes a static factory method on a class to create the bean. The object type returned from the invocation of the static factory method may be the same class or another class entirely.

Inner class names

If you want to configure a bean definition for a static nested class, you have to use the binary name of the nested class.

For example, if you have a class called Foo in the com.example package, and this Foo class has a static nested class called Bar, the value of the 'class' attribute on a bean definition would be…​

com.example.Foo$Bar

Notice the use of the $ character in the name to separate the nested class name from the outer class name.

Instantiation with a constructor

When you create a bean by the constructor approach, all normal classes are usable by and compatible with Spring. That is, the class being developed does not need to implement any specific interfaces or to be coded in a specific fashion. Simply specifying the bean class should suffice. However, depending on what type of IoC you use for that specific bean, you may need a default (empty) constructor.

The Spring IoC container can manage virtually any class you want it to manage; it is not limited to managing true JavaBeans. Most Spring users prefer actual JavaBeans with only a default (no-argument) constructor and appropriate setters and getters modeled after the properties in the container. You can also have more exotic non-bean-style classes in your container. If, for example, you need to use a legacy connection pool that absolutely does not adhere to the JavaBean specification, Spring can manage it as well.

With XML-based configuration metadata you can specify your bean class as follows:

<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>

For details about the mechanism for supplying arguments to the constructor (if required) and setting object instance properties after the object is constructed, see Injecting Dependencies.

Instantiation with a static factory method

When defining a bean that you create with a static factory method, you use the class attribute to specify the class containing the static factory method and an attribute named factory-method to specify the name of the factory method itself. You should be able to call this method (with optional arguments as described later) and return a live object, which subsequently is treated as if it had been created through a constructor. One use for such a bean definition is to call static factories in legacy code.

The following bean definition specifies that the bean will be created by calling a factory-method. The definition does not specify the type (class) of the returned object, only the class containing the factory method. In this example, the createInstance() method must be a static method.

<bean id="clientService"

class="examples.ClientService"

factory-method="createInstance"/>

public class ClientService {

private static ClientService clientService = new ClientService();

private ClientService() {}

public static ClientService createInstance() {

return clientService;

}

}

For details about the mechanism for supplying (optional) arguments to the factory method and setting object instance properties after the object is returned from the factory, see Dependencies and configuration in detail.

Instantiation using an instance factory method

Similar to instantiation through a static factory method, instantiation with an instance factory method invokes a non-static method of an existing bean from the container to create a new bean. To use this mechanism, leave the class attribute empty, and in the factory-bean attribute, specify the name of a bean in the current (or parent/ancestor) container that contains the instance method that is to be invoked to create the object. Set the name of the factory method itself with the factory-method attribute.

<!-- the factory bean, which contains a method called createInstance() -->

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

<!-- inject any dependencies required by this locator bean -->

</bean>

<!-- the bean to be created via the factory bean -->

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

public class DefaultServiceLocator {

private static ClientService clientService = new ClientServiceImpl();

public ClientService createClientServiceInstance() {

return clientService;

}

}

One factory class can also hold more than one factory method as shown here:

<bean id="serviceLocator" class="examples.DefaultServiceLocator">

<!-- inject any dependencies required by this locator bean -->

</bean>

<bean id="clientService"

factory-bean="serviceLocator"

factory-method="createClientServiceInstance"/>

<bean id="accountService"

factory-bean="serviceLocator"

factory-method="createAccountServiceInstance"/>

public class DefaultServiceLocator {

private static ClientService clientService = new ClientServiceImpl();

private static AccountService accountService = new AccountServiceImpl();

public ClientService createClientServiceInstance() {

return clientService;

}

public AccountService createAccountServiceInstance() {

return accountService;

}

}

This approach shows that the factory bean itself can be managed and configured through dependency injection (DI). See Dependencies and configuration in detail.

In Spring documentation, factory bean refers to a bean that is configured in the Spring container that will create objects through an instance or static factory method. By contrast, FactoryBean (notice the capitalization) refers to a Spring-specific FactoryBean .

1.4. Dependencies

A typical enterprise application does not consist of a single object (or bean in the Spring parlance). Even the simplest application has a few objects that work together to present what the end-user sees as a coherent application. This next section explains how you go from defining a number of bean definitions that stand alone to a fully realized application where objects collaborate to achieve a goal.

1.4.1. Dependency Injection

Dependency injection (DI) is a process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control (IoC), of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes, or the Service Locator pattern.

Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies, and does not know the location or class of the dependencies. As such, your classes become easier to test, in particular when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.

DI exists in two major variants, Constructor-based dependency injection and Setter-based dependency injection.

Constructor-based dependency injection

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency. Calling a static factory method with specific arguments to construct the bean is nearly equivalent, and this discussion treats arguments to a constructor and to a static factory method similarly. The following example shows a class that can only be dependency-injected with constructor injection. Notice that there is nothing special about this class, it is a POJO that has no dependencies on container specific interfaces, base classes or annotations.

public class SimpleMovieLister {

// the SimpleMovieLister has a dependency on a MovieFinder

private MovieFinder movieFinder;

// a constructor so that the Spring container can inject a MovieFinder

public SimpleMovieLister(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

// business logic that actually uses the injected MovieFinder is omitted...

}

Constructor argument resolution

Constructor argument resolution matching occurs using the argument’s type. If no potential ambiguity exists in the constructor arguments of a bean definition, then the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor when the bean is being instantiated. Consider the following class:

package x.y;

public class Foo {

public Foo(Bar bar, Baz baz) {

// ...

}

}

No potential ambiguity exists, assuming that Bar and Baz classes are not related by inheritance. Thus the following configuration works fine, and you do not need to specify the constructor argument indexes and/or types explicitly in the <constructor-arg/> element.

<beans>

<bean id="foo" class="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

</bean>

<bean id="bar" class="x.y.Bar"/>

<bean id="baz" class="x.y.Baz"/>

</beans>

When another bean is referenced, the type is known, and matching can occur (as was the case with the preceding example). When a simple type is used, such as <value>true</value>, Spring cannot determine the type of the value, and so cannot match by type without help. Consider the following class:

package examples;

public class ExampleBean {

// Number of years to calculate the Ultimate Answer

private int years;

// The Answer to Life, the Universe, and Everything

private String ultimateAnswer;

public ExampleBean(int years, String ultimateAnswer) {

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

}

Constructor argument type matching

In the preceding scenario, the container can use type matching with simple types if you explicitly specify the type of the constructor argument using the type attribute. For example:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg type="int" value="7500000"/>

<constructor-arg type="java.lang.String" value="42"/>

</bean>

Constructor argument index

Use the index attribute to specify explicitly the index of constructor arguments. For example:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg index="0" value="7500000"/>

<constructor-arg index="1" value="42"/>

</bean>

In addition to resolving the ambiguity of multiple simple values, specifying an index resolves ambiguity where a constructor has two arguments of the same type. Note that the index is 0 based.

Constructor argument name

You can also use the constructor parameter name for value disambiguation:

<bean id="exampleBean" class="examples.ExampleBean">

<constructor-arg name="years" value="7500000"/>

<constructor-arg name="ultimateAnswer" value="42"/>

</bean>

Keep in mind that to make this work out of the box your code must be compiled with the debug flag enabled so that Spring can look up the parameter name from the constructor. If you can’t compile your code with debug flag (or don’t want to) you can use @ConstructorProperties JDK annotation to explicitly name your constructor arguments. The sample class would then have to look as follows:

package examples;

public class ExampleBean {

// Fields omitted

@ConstructorProperties({"years", "ultimateAnswer"})

public ExampleBean(int years, String ultimateAnswer) {

this.years = years;

this.ultimateAnswer = ultimateAnswer;

}

}

Setter-based dependency injection

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

The following example shows a class that can only be dependency-injected using pure setter injection. This class is conventional Java. It is a POJO that has no dependencies on container specific interfaces, base classes or annotations.

public class SimpleMovieLister {

// the SimpleMovieLister has a dependency on the MovieFinder

private MovieFinder movieFinder;

// a setter method so that the Spring container can inject a MovieFinder

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

// business logic that actually uses the injected MovieFinder is omitted...

}

The ApplicationContext supports constructor-based and setter-based DI for the beans it manages. It also supports setter-based DI after some dependencies have already been injected through the constructor approach. You configure the dependencies in the form of a BeanDefinition, which you use in conjunction with PropertyEditor instances to convert properties from one format to another. However, most Spring users do not work with these classes directly (i.e., programmatically) but rather with XML bean definitions, annotated components (i.e., classes annotated with @Component, @Controller, etc.), or @Bean methods in Java-based @Configuration classes. These sources are then converted internally into instances of BeanDefinition and used to load an entire Spring IoC container instance.

Constructor-based or setter-based DI?

Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies. Note that use of the @Required annotation on a setter method can be used to make the property a required dependency.

The Spring team generally advocates constructor injection as it enables one to implement application components as immutable objects and to ensure that required dependencies are not null. Furthermore constructor-injected components are always returned to client (calling) code in a fully initialized state. As a side note, a large number of constructor arguments is a bad code smell, implying that the class likely has too many responsibilities and should be refactored to better address proper separation of concerns.

Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class. Otherwise, not-null checks must be performed everywhere the code uses the dependency. One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later. Management through JMX MBeans is therefore a compelling use case for setter injection.

Use the DI style that makes the most sense for a particular class. Sometimes, when dealing with third-party classes for which you do not have the source, the choice is made for you. For example, if a third-party class does not expose any setter methods, then constructor injection may be the only available form of DI.

Dependency resolution process

The container performs bean dependency resolution as follows:

The ApplicationContext is created and initialized with configuration metadata that describes all the beans. Configuration metadata can be specified via XML, Java code, or annotations.

For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method if you are using that instead of a normal constructor. These dependencies are provided to the bean, when the bean is actually created.

Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.

Each property or constructor argument which is a value is converted from its specified format to the actual type of that property or constructor argument. By default Spring can convert a value supplied in string format to all built-in types, such as int, long, String, boolean, etc.

The Spring container validates the configuration of each bean as the container is created. However, the bean properties themselves are not set until the bean is actually created. Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created. Scopes are defined in Bean scopes. Otherwise, the bean is created only when it is requested. Creation of a bean potentially causes a graph of beans to be created, as the bean’s dependencies and its dependencies' dependencies (and so on) are created and assigned. Note that resolution mismatches among those dependencies may show up late, i.e. on first creation of the affected bean.

Circular dependencies

If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.

For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a BeanCurrentlyInCreationException.

One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.

Unlike the typical case (with no circular dependencies), a circular dependency between bean A and bean B forces one of the beans to be injected into the other prior to being fully initialized itself (a classic chicken/egg scenario).

You can generally trust Spring to do the right thing. It detects configuration problems, such as references to non-existent beans and circular dependencies, at container load-time. Spring sets properties and resolves dependencies as late as possible, when the bean is actually created. This means that a Spring container which has loaded correctly can later generate an exception when you request an object if there is a problem creating that object or one of its dependencies. For example, the bean throws an exception as a result of a missing or invalid property. This potentially delayed visibility of some configuration issues is why ApplicationContext implementations by default pre-instantiate singleton beans. At the cost of some upfront time and memory to create these beans before they are actually needed, you discover configuration issues when the ApplicationContext is created, not later. You can still override this default behavior so that singleton beans will lazy-initialize, rather than be pre-instantiated.

If no circular dependencies exist, when one or more collaborating beans are being injected into a dependent bean, each collaborating bean is totally configured prior to being injected into the dependent bean. This means that if bean A has a dependency on bean B, the Spring IoC container completely configures bean B prior to invoking the setter method on bean A. In other words, the bean is instantiated (if not a pre-instantiated singleton), its dependencies are set, and the relevant lifecycle methods (such as a configured init method or the InitializingBean callback method) are invoked.

Examples of dependency injection

The following example uses XML-based configuration metadata for setter-based DI. A small part of a Spring XML configuration file specifies some bean definitions:

<bean id="exampleBean" class="examples.ExampleBean">

<!-- setter injection using the nested ref element -->

<property name="beanOne">

<ref bean="anotherExampleBean"/>

</property>

<!-- setter injection using the neater ref attribute -->

<property name="beanTwo" ref="yetAnotherBean"/>

<property name="integerProperty" value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

public class ExampleBean {

private AnotherBean beanOne;

private YetAnotherBean beanTwo;

private int i;

public void setBeanOne(AnotherBean beanOne) {

this.beanOne = beanOne;

}

public void setBeanTwo(YetAnotherBean beanTwo) {

this.beanTwo = beanTwo;

}

public void setIntegerProperty(int i) {

this.i = i;

}

}

In the preceding example, setters are declared to match against the properties specified in the XML file. The following example uses constructor-based DI:

<bean id="exampleBean" class="examples.ExampleBean">

<!-- constructor injection using the nested ref element -->

<constructor-arg>

<ref bean="anotherExampleBean"/>

</constructor-arg>

<!-- constructor injection using the neater ref attribute -->

<constructor-arg ref="yetAnotherBean"/>

<constructor-arg type="int" value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

public class ExampleBean {

private AnotherBean beanOne;

private YetAnotherBean beanTwo;

private int i;

public ExampleBean(

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {

this.beanOne = anotherBean;

this.beanTwo = yetAnotherBean;

this.i = i;

}

}

The constructor arguments specified in the bean definition will be used as arguments to the constructor of the ExampleBean.

Now consider a variant of this example, where instead of using a constructor, Spring is told to call a static factory method to return an instance of the object:

<bean id="exampleBean" class="examples.ExampleBean" factory-method="createInstance">

<constructor-arg ref="anotherExampleBean"/>

<constructor-arg ref="yetAnotherBean"/>

<constructor-arg value="1"/>

</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>

<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

public class ExampleBean {

// a private constructor

private ExampleBean(...) {

...

}

// a static factory method; the arguments to this method can be

// considered the dependencies of the bean that is returned,

// regardless of how those arguments are actually used.

public static ExampleBean createInstance (

AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {

ExampleBean eb = new ExampleBean (...);

// some other operations...

return eb;

}

}

Arguments to the static factory method are supplied via <constructor-arg/> elements, exactly the same as if a constructor had actually been used. The type of the class being returned by the factory method does not have to be of the same type as the class that contains the static factory method, although in this example it is. An instance (non-static) factory method would be used in an essentially identical fashion (aside from the use of the factory-bean attribute instead of the class attribute), so details will not be discussed here.

1.4.2. Dependencies and configuration in detail

As mentioned in the previous section, you can define bean properties and constructor arguments as references to other managed beans (collaborators), or as values defined inline. Spring’s XML-based configuration metadata supports sub-element types within its <property/> and <constructor-arg/> elements for this purpose.

Straight values (primitives, Strings, and so on)

The value attribute of the <property/> element specifies a property or constructor argument as a human-readable string representation. Spring’s conversion service is used to convert these values from a String to the actual type of the property or argument.

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<!-- results in a setDriverClassName(String) call -->

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql://localhost:3306/mydb"/>

<property name="username" value="root"/>

<property name="password" value="masterkaoli"/>

</bean>

The following example uses the p-namespace for even more succinct XML configuration.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close"

p:driverClassName="com.mysql.jdbc.Driver"

p:url="jdbc:mysql://localhost:3306/mydb"

p:username="root"

p:password="masterkaoli"/>

</beans>

The preceding XML is more succinct; however, typos are discovered at runtime rather than design time, unless you use an IDE such as IntelliJ IDEA or the Spring Tool Suite (STS) that support automatic property completion when you create bean definitions. Such IDE assistance is highly recommended.

You can also configure a java.util.Properties instance as:

<bean id="mappings"

class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<!-- typed as a java.util.Properties -->

<property name="properties">

<value>

jdbc.driver.className=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mydb

</value>

</property>

</bean>

The Spring container converts the text inside the <value/> element into a java.util.Properties instance by using the JavaBeans PropertyEditor mechanism. This is a nice shortcut, and is one of a few places where the Spring team do favor the use of the nested <value/> element over the value attribute style.

The idref element

The idref element is simply an error-proof way to pass the id (string value - not a reference) of another bean in the container to a <constructor-arg/> or <property/> element.

<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">

<property name="targetName">

<idref bean="theTargetBean"/>

</property>

</bean>

The above bean definition snippet is exactly equivalent (at runtime) to the following snippet:

<bean id="theTargetBean" class="..." />

<bean id="client" class="...">

<property name="targetName" value="theTargetBean"/>

</bean>

The first form is preferable to the second, because using the idref tag allows the container to validate at deployment time that the referenced, named bean actually exists. In the second variation, no validation is performed on the value that is passed to the targetName property of the client bean. Typos are only discovered (with most likely fatal results) when the client bean is actually instantiated. If the client bean is a prototype bean, this typo and the resulting exception may only be discovered long after the container is deployed.

The local attribute on the idref element is no longer supported in the 4.0 beans xsd since it does not provide value over a regular bean reference anymore. Simply change your existing idref local references to idref bean when upgrading to the 4.0 schema.

A common place (at least in versions earlier than Spring 2.0) where the <idref/> element brings value is in the configuration of AOP interceptors in a ProxyFactoryBean bean definition. Using <idref/> elements when you specify the interceptor names prevents you from misspelling an interceptor id.

References to other beans (collaborators)

The ref element is the final element inside a <constructor-arg/> or <property/> definition element. Here you set the value of the specified property of a bean to be a reference to another bean (a collaborator) managed by the container. The referenced bean is a dependency of the bean whose property will be set, and it is initialized on demand as needed before the property is set. (If the collaborator is a singleton bean, it may be initialized already by the container.) All references are ultimately a reference to another object. Scoping and validation depend on whether you specify the id/name of the other object through the bean, local, or parent attributes.

Specifying the target bean through the bean attribute of the <ref/> tag is the most general form, and allows creation of a reference to any bean in the same container or parent container, regardless of whether it is in the same XML file. The value of the bean attribute may be the same as the id attribute of the target bean, or as one of the values in the name attribute of the target bean.

<ref bean="someBean"/>

Specifying the target bean through the parent attribute creates a reference to a bean that is in a parent container of the current container. The value of the parent attribute may be the same as either the id attribute of the target bean, or one of the values in the name attribute of the target bean, and the target bean must be in a parent container of the current one. You use this bean reference variant mainly when you have a hierarchy of containers and you want to wrap an existing bean in a parent container with a proxy that will have the same name as the parent bean.

<!-- in the parent context -->

<bean id="accountService" class="com.foo.SimpleAccountService">

<!-- insert dependencies as required as here -->

</bean>

<!-- in the child (descendant) context -->

<bean id="accountService" <!-- bean name is the same as the parent bean -->

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target">

<ref parent="accountService"/> <!-- notice how we refer to the parent bean -->

</property>

<!-- insert other configuration and dependencies as required here -->

</bean>

The local attribute on the ref element is no longer supported in the 4.0 beans xsd since it does not provide value over a regular bean reference anymore. Simply change your existing ref local references to ref bean when upgrading to the 4.0 schema.

Inner beans

A <bean/> element inside the <property/> or <constructor-arg/> elements defines a so-called inner bean.

<bean id="outer" class="...">

<!-- instead of using a reference to a target bean, simply define the target bean inline -->

<property name="target">

<bean class="com.example.Person"> <!-- this is the inner bean -->

<property name="name" value="Fiona Apple"/>

<property name="age" value="25"/>

</bean>

</property>

</bean>

An inner bean definition does not require a defined id or name; if specified, the container does not use such a value as an identifier. The container also ignores the scope flag on creation: Inner beans are always anonymous and they are always created with the outer bean. It is not possible to inject inner beans into collaborating beans other than into the enclosing bean or to access them independently.

As a corner case, it is possible to receive destruction callbacks from a custom scope, e.g. for a request-scoped inner bean contained within a singleton bean: The creation of the inner bean instance will be tied to its containing bean, but destruction callbacks allow it to participate in the request scope’s lifecycle. This is not a common scenario; inner beans typically simply share their containing bean’s scope.

Collections

In the <list/>, <set/>, <map/>, and <props/> elements, you set the properties and arguments of the Java Collection types List, Set, Map, and Properties, respectively.

<bean id="moreComplexObject" class="example.ComplexObject">

<!-- results in a setAdminEmails(java.util.Properties) call -->

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.org</prop>

<prop key="support">support@example.org</prop>

<prop key="development">development@example.org</prop>

</props>

</property>

<!-- results in a setSomeList(java.util.List) call -->

<property name="someList">

<list>

<value>a list element followed by a reference</value>

<ref bean="myDataSource" />

</list>

</property>

<!-- results in a setSomeMap(java.util.Map) call -->

<property name="someMap">

<map>

<entry key="an entry" value="just some string"/>

<entry key ="a ref" value-ref="myDataSource"/>

</map>

</property>

<!-- results in a setSomeSet(java.util.Set) call -->

<property name="someSet">

<set>

<value>just some string</value>

<ref bean="myDataSource" />

</set>

</property>

</bean>

The value of a map key or value, or a set value, can also again be any of the following elements:

bean | ref | idref | list | set | map | props | value | null

Collection merging

The Spring container also supports the merging of collections. An application developer can define a parent-style <list/>, <map/>, <set/> or <props/> element, and have child-style <list/>, <map/>, <set/> or <props/> elements inherit and override values from the parent collection. That is, the child collection’s values are the result of merging the elements of the parent and child collections, with the child’s collection elements overriding values specified in the parent collection.

This section on merging discusses the parent-child bean mechanism. Readers unfamiliar with parent and child bean definitions may wish to read the relevant section before continuing.

The following example demonstrates collection merging:

<beans>

<bean id="parent" abstract="true" class="example.ComplexObject">

<property name="adminEmails">

<props>

<prop key="administrator">administrator@example.com</prop>

<prop key="support">support@example.com</prop>

</props>

</property>

</bean>

<bean id="child" parent="parent">

<property name="adminEmails">

<!-- the merge is specified on the child collection definition -->

<props merge="true">

<prop key="sales">sales@example.com</prop>

<prop key="support">support@example.co.uk</prop>

</props>

</property>

</bean>

<beans>

Notice the use of the merge=true attribute on the <props/> element of the adminEmails property of the child bean definition. When the child bean is resolved and instantiated by the container, the resulting instance has an adminEmails Properties collection that contains the result of the merging of the child’s adminEmails collection with the parent’s adminEmails collection.

administrator=administrator@example.com

sales=sales@example.com

support=support@example.co.uk

The child Properties collection’s value set inherits all property elements from the parent <props/>, and the child’s value for the support value overrides the value in the parent collection.

This merging behavior applies similarly to the <list/>, <map/>, and <set/> collection types. In the specific case of the <list/> element, the semantics associated with the List collection type, that is, the notion of an ordered collection of values, is maintained; the parent’s values precede all of the child list’s values. In the case of the Map, Set, and Properties collection types, no ordering exists. Hence no ordering semantics are in effect for the collection types that underlie the associated Map, Set, and Properties implementation types that the container uses internally.

Limitations of collection merging

You cannot merge different collection types (such as a Map and a List), and if you do attempt to do so an appropriate Exception is thrown. The merge attribute must be specified on the lower, inherited, child definition; specifying the merge attribute on a parent collection definition is redundant and will not result in the desired merging.

Strongly-typed collection

With the introduction of generic types in Java 5, you can use strongly typed collections. That is, it is possible to declare a Collection type such that it can only contain String elements (for example). If you are using Spring to dependency-inject a strongly-typed Collection into a bean, you can take advantage of Spring’s type-conversion support such that the elements of your strongly-typed Collection instances are converted to the appropriate type prior to being added to the Collection.

public class Foo {

private Map<String, Float> accounts;

public void setAccounts(Map<String, Float> accounts) {

this.accounts = accounts;

}

}

<beans>

<bean id="foo" class="x.y.Foo">

<property name="accounts">

<map>

<entry key="one" value="9.99"/>

<entry key="two" value="2.75"/>

<entry key="six" value="3.99"/>

</map>

</property>

</bean>

</beans>

When the accounts property of the foo bean is prepared for injection, the generics information about the element type of the strongly-typed Map<String, Float> is available by reflection. Thus Spring’s type conversion infrastructure recognizes the various value elements as being of type Float, and the string values 9.99, 2.75, and 3.99 are converted into an actual Float type.

Null and empty string values

Spring treats empty arguments for properties and the like as empty Strings. The following XML-based configuration metadata snippet sets the email property to the empty String value ("").

<bean class="ExampleBean">

<property name="email" value=""/>

</bean>

The preceding example is equivalent to the following Java code:

exampleBean.setEmail("");

The <null/> element handles null values. For example:

<bean class="ExampleBean">

<property name="email">

<null/>

</property>

</bean>

The above configuration is equivalent to the following Java code:

exampleBean.setEmail(null);

XML shortcut with the p-namespace

The p-namespace enables you to use the bean element’s attributes, instead of nested <property/> elements, to describe your property values and/or collaborating beans.

Spring supports extensible configuration formats with namespaces, which are based on an XML Schema definition. The beans configuration format discussed in this chapter is defined in an XML Schema document. However, the p-namespace is not defined in an XSD file and exists only in the core of Spring.

The following example shows two XML snippets that resolve to the same result: The first uses standard XML format and the second uses the p-namespace.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean name="classic" class="com.example.ExampleBean">

<property name="email" value="foo@bar.com"/>

</bean>

<bean name="p-namespace" class="com.example.ExampleBean"

p:email="foo@bar.com"/>

</beans>

The example shows an attribute in the p-namespace called email in the bean definition. This tells Spring to include a property declaration. As previously mentioned, the p-namespace does not have a schema definition, so you can set the name of the attribute to the property name.

This next example includes two more bean definitions that both have a reference to another bean:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean name="john-classic" class="com.example.Person">

<property name="name" value="John Doe"/>

<property name="spouse" ref="jane"/>

</bean>

<bean name="john-modern"

class="com.example.Person"

p:name="John Doe"

p:spouse-ref="jane"/>

<bean name="jane" class="com.example.Person">

<property name="name" value="Jane Doe"/>

</bean>

</beans>

As you can see, this example includes not only a property value using the p-namespace, but also uses a special format to declare property references. Whereas the first bean definition uses <property name="spouse" ref="jane"/> to create a reference from bean john to bean jane, the second bean definition uses p:spouse-ref="jane" as an attribute to do the exact same thing. In this case spouse is the property name, whereas the -ref part indicates that this is not a straight value but rather a reference to another bean.

The p-namespace is not as flexible as the standard XML format. For example, the format for declaring property references clashes with properties that end in Ref, whereas the standard XML format does not. We recommend that you choose your approach carefully and communicate this to your team members, to avoid producing XML documents that use all three approaches at the same time.

XML shortcut with the c-namespace

Similar to the XML shortcut with the p-namespace, the c-namespace, newly introduced in Spring 3.1, allows usage of inlined attributes for configuring the constructor arguments rather then nested constructor-arg elements.

Let’s review the examples from Constructor-based dependency injection with the c: namespace:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:c="http://www.springframework.org/schema/c"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<bean id="bar" class="x.y.Bar"/>

<bean id="baz" class="x.y.Baz"/>

<!-- traditional declaration -->

<bean id="foo" class="x.y.Foo">

<constructor-arg ref="bar"/>

<constructor-arg ref="baz"/>

<constructor-arg value="foo@bar.com"/>

</bean>

<!-- c-namespace declaration -->

<bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email="foo@bar.com"/>

</beans>

The c: namespace uses the same conventions as the p: one (trailing -ref for bean references) for setting the constructor arguments by their names. And just as well, it needs to be declared even though it is not defined in an XSD schema (but it exists inside the Spring core).

For the rare cases where the constructor argument names are not available (usually if the bytecode was compiled without debugging information), one can use fallback to the argument indexes:

<!-- c-namespace index declaration -->

<bean id="foo" class="x.y.Foo" c:\_0-ref="bar" c:\_1-ref="baz"/>

Due to the XML grammar, the index notation requires the presence of the leading \_ as XML attribute names cannot start with a number (even though some IDE allow it).

In practice, the constructor resolution mechanism is quite efficient in matching arguments so unless one really needs to, we recommend using the name notation through-out your configuration.

Compound property names

You can use compound or nested property names when you set bean properties, as long as all components of the path except the final property name are not null. Consider the following bean definition.

<bean id="foo" class="foo.Bar">

<property name="fred.bob.sammy" value="123" />

</bean>

The foo bean has a fred property, which has a bob property, which has a sammy property, and that final sammy property is being set to the value 123. In order for this to work, the fred property of foo, and the bob property of fred must not be null after the bean is constructed, or a NullPointerException is thrown.

1.4.3. Using depends-on

If a bean is a dependency of another that usually means that one bean is set as a property of another. Typically you accomplish this with the <ref/> element in XML-based configuration metadata. However, sometimes dependencies between beans are less direct; for example, a static initializer in a class needs to be triggered, such as database driver registration. The depends-on attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized. The following example uses the depends-on attribute to express a dependency on a single bean:

<bean id="beanOne" class="ExampleBean" depends-on="manager"/>

<bean id="manager" class="ManagerBean" />

To express a dependency on multiple beans, supply a list of bean names as the value of the depends-on attribute, with commas, whitespace and semicolons, used as valid delimiters:

<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">

<property name="manager" ref="manager" />

</bean>

<bean id="manager" class="ManagerBean" />

<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />

The depends-on attribute in the bean definition can specify both an initialization time dependency and, in the case of singleton beans only, a corresponding destroy time dependency. Dependent beans that define a depends-on relationship with a given bean are destroyed first, prior to the given bean itself being destroyed. Thus depends-on can also control shutdown order.

1.4.4. Lazy-initialized beans

By default, ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process. Generally, this pre-instantiation is desirable, because errors in the configuration or surrounding environment are discovered immediately, as opposed to hours or even days later. When this behavior is not desirable, you can prevent pre-instantiation of a singleton bean by marking the bean definition as lazy-initialized. A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

In XML, this behavior is controlled by the lazy-init attribute on the <bean/> element; for example:

<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>

When the preceding configuration is consumed by an ApplicationContext, the bean named lazy is not eagerly pre-instantiated when the ApplicationContext is starting up, whereas the not.lazy bean is eagerly pre-instantiated.

However, when a lazy-initialized bean is a dependency of a singleton bean that is not lazy-initialized, the ApplicationContext creates the lazy-initialized bean at startup, because it must satisfy the singleton’s dependencies. The lazy-initialized bean is injected into a singleton bean elsewhere that is not lazy-initialized.

You can also control lazy-initialization at the container level by using the default-lazy-init attribute on the <beans/> element; for example:

<beans default-lazy-init="true">

<!-- no beans will be pre-instantiated... -->

</beans>

1.4.5. Autowiring collaborators

The Spring container can autowire relationships between collaborating beans. You can allow Spring to resolve collaborators (other beans) automatically for your bean by inspecting the contents of the ApplicationContext. Autowiring has the following advantages:

Autowiring can significantly reduce the need to specify properties or constructor arguments. (Other mechanisms such as a bean template discussed elsewhere in this chapter are also valuable in this regard.)

Autowiring can update a configuration as your objects evolve. For example, if you need to add a dependency to a class, that dependency can be satisfied automatically without you needing to modify the configuration. Thus autowiring can be especially useful during development, without negating the option of switching to explicit wiring when the code base becomes more stable.

When using XML-based configuration metadata [2], you specify autowire mode for a bean definition with the autowire attribute of the <bean/> element. The autowiring functionality has four modes. You specify autowiring per bean and thus can choose which ones to autowire.

Table 2. Autowiring modes

Mode Explanation

no

(Default) No autowiring. Bean references must be defined via a ref element. Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity. To some extent, it documents the structure of a system.

byName

Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired. For example, if a bean definition is set to autowire by name, and it contains a master property (that is, it has a setMaster(..) method), Spring looks for a bean definition named master, and uses it to set the property.

byType

Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use byType autowiring for that bean. If there are no matching beans, nothing happens; the property is not set.

constructor

Analogous to byType, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

With byType or constructor autowiring mode, you can wire arrays and typed-collections. In such cases all autowire candidates within the container that match the expected type are provided to satisfy the dependency. You can autowire strongly-typed Maps if the expected key type is String. An autowired Maps values will consist of all bean instances that match the expected type, and the Maps keys will contain the corresponding bean names.

You can combine autowire behavior with dependency checking, which is performed after autowiring completes.

Limitations and disadvantages of autowiring

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions.

Consider the limitations and disadvantages of autowiring:

Explicit dependencies in property and constructor-arg settings always override autowiring. You cannot autowire so-called simple properties such as primitives, Strings, and Classes (and arrays of such simple properties). This limitation is by-design.

Autowiring is less exact than explicit wiring. Although, as noted in the above table, Spring is careful to avoid guessing in case of ambiguity that might have unexpected results, the relationships between your Spring-managed objects are no longer documented explicitly.

Wiring information may not be available to tools that may generate documentation from a Spring container.

Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or Maps, this is not necessarily a problem. However for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

In the latter scenario, you have several options:

Abandon autowiring in favor of explicit wiring.

Avoid autowiring for a bean definition by setting its autowire-candidate attributes to false as described in the next section.

Designate a single bean definition as the primary candidate by setting the primary attribute of its <bean/> element to true.

Implement the more fine-grained control available with annotation-based configuration, as described in Annotation-based container configuration.

Excluding a bean from autowiring

On a per-bean basis, you can exclude a bean from autowiring. In Spring’s XML format, set the autowire-candidate attribute of the <bean/> element to false; the container makes that specific bean definition unavailable to the autowiring infrastructure (including annotation style configurations such as @Autowired).

The autowire-candidate attribute is designed to only affect type-based autowiring. It does not affect explicit references by name, which will get resolved even if the specified bean is not marked as an autowire candidate. As a consequence, autowiring by name will nevertheless inject a bean if the name matches.

You can also limit autowire candidates based on pattern-matching against bean names. The top-level <beans/> element accepts one or more patterns within its default-autowire-candidates attribute. For example, to limit autowire candidate status to any bean whose name ends with Repository, provide a value of \*Repository. To provide multiple patterns, define them in a comma-separated list. An explicit value of true or false for a bean definitions autowire-candidate attribute always takes precedence, and for such beans, the pattern matching rules do not apply.

These techniques are useful for beans that you never want to be injected into other beans by autowiring. It does not mean that an excluded bean cannot itself be configured using autowiring. Rather, the bean itself is not a candidate for autowiring other beans.

1.4.6. Method injection

In most application scenarios, most beans in the container are singletons. When a singleton bean needs to collaborate with another singleton bean, or a non-singleton bean needs to collaborate with another non-singleton bean, you typically handle the dependency by defining one bean as a property of the other. A problem arises when the bean lifecycles are different. Suppose singleton bean A needs to use non-singleton (prototype) bean B, perhaps on each method invocation on A. The container only creates the singleton bean A once, and thus only gets one opportunity to set the properties. The container cannot provide bean A with a new instance of bean B every time one is needed.

A solution is to forego some inversion of control. You can make bean A aware of the container by implementing the ApplicationContextAware interface, and by making a getBean("B") call to the container ask for (a typically new) bean B instance every time bean A needs it. The following is an example of this approach:

// a class that uses a stateful Command-style class to perform some processing

package fiona.apple;

// Spring-API imports

import org.springframework.beans.BeansException;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

public class CommandManager implements ApplicationContextAware {

private ApplicationContext applicationContext;

public Object process(Map commandState) {

// grab a new instance of the appropriate Command

Command command = createCommand();

// set the state on the (hopefully brand new) Command instance

command.setState(commandState);

return command.execute();

}

protected Command createCommand() {

// notice the Spring API dependency!

return this.applicationContext.getBean("command", Command.class);

}

public void setApplicationContext(

ApplicationContext applicationContext) throws BeansException {

this.applicationContext = applicationContext;

}

}

The preceding is not desirable, because the business code is aware of and coupled to the Spring Framework. Method Injection, a somewhat advanced feature of the Spring IoC container, allows this use case to be handled in a clean fashion.

You can read more about the motivation for Method Injection in this blog entry.

Lookup method injection

Lookup method injection is the ability of the container to override methods on container managed beans, to return the lookup result for another named bean in the container. The lookup typically involves a prototype bean as in the scenario described in the preceding section. The Spring Framework implements this method injection by using bytecode generation from the CGLIB library to generate dynamically a subclass that overrides the method.

For this dynamic subclassing to work, the class that the Spring bean container will subclass cannot be final, and the method to be overridden cannot be final either.

Unit-testing a class that has an abstract method requires you to subclass the class yourself and to supply a stub implementation of the abstract method.

Concrete methods are also necessary for component scanning which requires concrete classes to pick up.

A further key limitation is that lookup methods won’t work with factory methods and in particular not with @Bean methods in configuration classes, since the container is not in charge of creating the instance in that case and therefore cannot create a runtime-generated subclass on the fly.

Looking at the CommandManager class in the previous code snippet, you see that the Spring container will dynamically override the implementation of the createCommand() method. Your CommandManager class will not have any Spring dependencies, as can be seen in the reworked example:

package fiona.apple;

// no more Spring imports!

public abstract class CommandManager {

public Object process(Object commandState) {

// grab a new instance of the appropriate Command interface

Command command = createCommand();

// set the state on the (hopefully brand new) Command instance

command.setState(commandState);

return command.execute();

}

// okay... but where is the implementation of this method?

protected abstract Command createCommand();

}

In the client class containing the method to be injected (the CommandManager in this case), the method to be injected requires a signature of the following form:

<public|protected> [abstract] <return-type> theMethodName(no-arguments);

If the method is abstract, the dynamically-generated subclass implements the method. Otherwise, the dynamically-generated subclass overrides the concrete method defined in the original class. For example:

<!-- a stateful bean deployed as a prototype (non-singleton) -->

<bean id="myCommand" class="fiona.apple.AsyncCommand" scope="prototype">

<!-- inject dependencies here as required -->

</bean>

<!-- commandProcessor uses statefulCommandHelper -->

<bean id="commandManager" class="fiona.apple.CommandManager">

<lookup-method name="createCommand" bean="myCommand"/>

</bean>

The bean identified as commandManager calls its own method createCommand() whenever it needs a new instance of the myCommand bean. You must be careful to deploy the myCommand bean as a prototype, if that is actually what is needed. If it is as a singleton, the same instance of the myCommand bean is returned each time.

Alternatively, within the annotation-based component model, you may declare a lookup method through the @Lookup annotation:

public abstract class CommandManager {

public Object process(Object commandState) {

Command command = createCommand();

command.setState(commandState);

return command.execute();

}

@Lookup("myCommand")

protected abstract Command createCommand();

}

Or, more idiomatically, you may rely on the target bean getting resolved against the declared return type of the lookup method:

public abstract class CommandManager {

public Object process(Object commandState) {

MyCommand command = createCommand();

command.setState(commandState);

return command.execute();

}

@Lookup

protected abstract MyCommand createCommand();

}

Note that you will typically declare such annotated lookup methods with a concrete stub implementation, in order for them to be compatible with Spring’s component scanning rules where abstract classes get ignored by default. This limitation does not apply in case of explicitly registered or explicitly imported bean classes.

Another way of accessing differently scoped target beans is an ObjectFactory/ Provider injection point. Check out Scoped beans as dependencies.

The interested reader may also find the ServiceLocatorFactoryBean (in the org.springframework.beans.factory.config package) to be of use.

Arbitrary method replacement

A less useful form of method injection than lookup method injection is the ability to replace arbitrary methods in a managed bean with another method implementation. Users may safely skip the rest of this section until the functionality is actually needed.

With XML-based configuration metadata, you can use the replaced-method element to replace an existing method implementation with another, for a deployed bean. Consider the following class, with a method computeValue, which we want to override:

public class MyValueCalculator {

public String computeValue(String input) {

// some real code...

}

// some other methods...

}

A class implementing the org.springframework.beans.factory.support.MethodReplacer interface provides the new method definition.

/\*\*

\* meant to be used to override the existing computeValue(String)

\* implementation in MyValueCalculator

\*/

public class ReplacementComputeValue implements MethodReplacer {

public Object reimplement(Object o, Method m, Object[] args) throws Throwable {

// get the input value, work with it, and return a computed result

String input = (String) args[0];

...

return ...;

}

}

The bean definition to deploy the original class and specify the method override would look like this:

<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">

<!-- arbitrary method replacement -->

<replaced-method name="computeValue" replacer="replacementComputeValue">

<arg-type>String</arg-type>

</replaced-method>

</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>

You can use one or more contained <arg-type/> elements within the <replaced-method/> element to indicate the method signature of the method being overridden. The signature for the arguments is necessary only if the method is overloaded and multiple variants exist within the class. For convenience, the type string for an argument may be a substring of the fully qualified type name. For example, the following all match java.lang.String:

java.lang.String

String

Str

Because the number of arguments is often enough to distinguish between each possible choice, this shortcut can save a lot of typing, by allowing you to type only the shortest string that will match an argument type.

1.5. Bean scopes

When you create a bean definition, you create a recipe for creating actual instances of the class defined by that bean definition. The idea that a bean definition is a recipe is important, because it means that, as with a class, you can create many object instances from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition, but also the scope of the objects created from a particular bean definition. This approach is powerful and flexible in that you can choose the scope of the objects you create through configuration instead of having to bake in the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports six scopes, four of which are available only if you use a web-aware ApplicationContext.

The following scopes are supported out of the box. You can also create a custom scope.

Table 3. Bean scopes

Scope Description

singleton

(Default) Scopes a single bean definition to a single object instance per Spring IoC container.

prototype

Scopes a single bean definition to any number of object instances.

request

Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.

session

Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.

application

Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.

websocket

Scopes a single bean definition to the lifecycle of a WebSocket. Only valid in the context of a web-aware Spring ApplicationContext.

As of Spring 3.0, a thread scope is available, but is not registered by default. For more information, see the documentation for SimpleThreadScope. For instructions on how to register this or any other custom scope, see Using a custom scope.

1.5.1. The singleton scope

Only one shared instance of a singleton bean is managed, and all requests for beans with an id or ids matching that bean definition result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

singleton

Spring’s concept of a singleton bean differs from the Singleton pattern as defined in the Gang of Four (GoF) patterns book. The GoF Singleton hard-codes the scope of an object such that one and only one instance of a particular class is created per ClassLoader. The scope of the Spring singleton is best described as per container and per bean. This means that if you define one bean for a particular class in a single Spring container, then the Spring container creates one and only one instance of the class defined by that bean definition. The singleton scope is the default scope in Spring. To define a bean as a singleton in XML, you would write, for example:

<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the default) -->

<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

1.5.2. The prototype scope

The non-singleton, prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made. That is, the bean is injected into another bean or you request it through a getBean() method call on the container. As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans.

The following diagram illustrates the Spring prototype scope. A data access object (DAO) is not typically configured as a prototype, because a typical DAO does not hold any conversational state; it was just easier for this author to reuse the core of the singleton diagram.

prototype

The following example defines a bean as a prototype in XML:

<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, and otherwise assembles a prototype object, and hands it to the client, with no further record of that prototype instance. Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured destruction lifecycle callbacks are not called. The client code must clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding. To get the Spring container to release resources held by prototype-scoped beans, try using a custom bean post-processor, which holds a reference to beans that need to be cleaned up.

In some respects, the Spring container’s role in regard to a prototype-scoped bean is a replacement for the Java new operator. All lifecycle management past that point must be handled by the client. (For details on the lifecycle of a bean in the Spring container, see Lifecycle callbacks.)

1.5.3. Singleton beans with prototype-bean dependencies

When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time. Thus if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.

However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only once, when the Spring container is instantiating the singleton bean and resolving and injecting its dependencies. If you need a new instance of a prototype bean at runtime more than once, see Method injection

1.5.4. Request, session, application, and WebSocket scopes

The request, session, application, and websocket scopes are only available if you use a web-aware Spring ApplicationContext implementation (such as XmlWebApplicationContext). If you use these scopes with regular Spring IoC containers such as the ClassPathXmlApplicationContext, an IllegalStateException will be thrown complaining about an unknown bean scope.

Initial web configuration

To support the scoping of beans at the request, session, application, and websocket levels (web-scoped beans), some minor initial configuration is required before you define your beans. (This initial setup is not required for the standard scopes, singleton and prototype.)

How you accomplish this initial setup depends on your particular Servlet environment.

If you access scoped beans within Spring Web MVC, in effect, within a request that is processed by the Spring DispatcherServlet, then no special setup is necessary: DispatcherServlet already exposes all relevant state.

If you use a Servlet 2.5 web container, with requests processed outside of Spring’s DispatcherServlet (for example, when using JSF or Struts), you need to register the org.springframework.web.context.request.RequestContextListener ServletRequestListener. For Servlet 3.0+, this can be done programmatically via the WebApplicationInitializer interface. Alternatively, or for older containers, add the following declaration to your web application’s web.xml file:

<web-app>

...

<listener>

<listener-class>

org.springframework.web.context.request.RequestContextListener

</listener-class>

</listener>

...

</web-app>

Alternatively, if there are issues with your listener setup, consider using Spring’s RequestContextFilter. The filter mapping depends on the surrounding web application configuration, so you have to change it as appropriate.

<web-app>

...

<filter>

<filter-name>requestContextFilter</filter-name>

<filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>requestContextFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

...

</web-app>

DispatcherServlet, RequestContextListener, and RequestContextFilter all do exactly the same thing, namely bind the HTTP request object to the Thread that is servicing that request. This makes beans that are request- and session-scoped available further down the call chain.

Request scope

Consider the following XML configuration for a bean definition:

<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>

The Spring container creates a new instance of the LoginAction bean by using the loginAction bean definition for each and every HTTP request. That is, the loginAction bean is scoped at the HTTP request level. You can change the internal state of the instance that is created as much as you want, because other instances created from the same loginAction bean definition will not see these changes in state; they are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.

When using annotation-driven components or Java Config, the @RequestScope annotation can be used to assign a component to the request scope.

@RequestScope

@Component

public class LoginAction {

// ...

}

Session scope

Consider the following XML configuration for a bean definition:

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

The Spring container creates a new instance of the UserPreferences bean by using the userPreferences bean definition for the lifetime of a single HTTP Session. In other words, the userPreferences bean is effectively scoped at the HTTP Session level. As with request-scoped beans, you can change the internal state of the instance that is created as much as you want, knowing that other HTTP Session instances that are also using instances created from the same userPreferences bean definition do not see these changes in state, because they are particular to an individual HTTP Session. When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session is also discarded.

When using annotation-driven components or Java Config, the @SessionScope annotation can be used to assign a component to the session scope.

@SessionScope

@Component

public class UserPreferences {

// ...

}

Application scope

Consider the following XML configuration for a bean definition:

<bean id="appPreferences" class="com.foo.AppPreferences" scope="application"/>

The Spring container creates a new instance of the AppPreferences bean by using the appPreferences bean definition once for the entire web application. That is, the appPreferences bean is scoped at the ServletContext level, stored as a regular ServletContext attribute. This is somewhat similar to a Spring singleton bean but differs in two important ways: It is a singleton per ServletContext, not per Spring 'ApplicationContext' (for which there may be several in any given web application), and it is actually exposed and therefore visible as a ServletContext attribute.

When using annotation-driven components or Java Config, the @ApplicationScope annotation can be used to assign a component to the application scope.

@ApplicationScope

@Component

public class AppPreferences {

// ...

}

Scoped beans as dependencies

The Spring IoC container manages not only the instantiation of your objects (beans), but also the wiring up of collaborators (or dependencies). If you want to inject (for example) an HTTP request scoped bean into another bean of a longer-lived scope, you may choose to inject an AOP proxy in place of the scoped bean. That is, you need to inject a proxy object that exposes the same public interface as the scoped object but that can also retrieve the real target object from the relevant scope (such as an HTTP request) and delegate method calls onto the real object.

You may also use <aop:scoped-proxy/> between beans that are scoped as singleton, with the reference then going through an intermediate proxy that is serializable and therefore able to re-obtain the target singleton bean on deserialization.

When declaring <aop:scoped-proxy/> against a bean of scope prototype, every method call on the shared proxy will lead to the creation of a new target instance which the call is then being forwarded to.

Also, scoped proxies are not the only way to access beans from shorter scopes in a lifecycle-safe fashion. You may also simply declare your injection point (i.e. the constructor/setter argument or autowired field) as ObjectFactory<MyTargetBean>, allowing for a getObject() call to retrieve the current instance on demand every time it is needed - without holding on to the instance or storing it separately.

As an extended variant, you may declare ObjectProvider<MyTargetBean> which delivers several additional access variants, including getIfAvailable and getIfUnique.

The JSR-330 variant of this is called Provider, used with a Provider<MyTargetBean> declaration and a corresponding get() call for every retrieval attempt. See here for more details on JSR-330 overall.

The configuration in the following example is only one line, but it is important to understand the "why" as well as the "how" behind it.

<?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:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- an HTTP Session-scoped bean exposed as a proxy -->

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

<!-- instructs the container to proxy the surrounding bean -->

<aop:scoped-proxy/>

</bean>

<!-- a singleton-scoped bean injected with a proxy to the above bean -->

<bean id="userService" class="com.foo.SimpleUserService">

<!-- a reference to the proxied userPreferences bean -->

<property name="userPreferences" ref="userPreferences"/>

</bean>

</beans>

To create such a proxy, you insert a child <aop:scoped-proxy/> element into a scoped bean definition (see Choosing the type of proxy to create and XML Schema-based configuration). Why do definitions of beans scoped at the request, session and custom-scope levels require the <aop:scoped-proxy/> element? Let’s examine the following singleton bean definition and contrast it with what you need to define for the aforementioned scopes (note that the following userPreferences bean definition as it stands is incomplete).

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

<bean id="userManager" class="com.foo.UserManager">

<property name="userPreferences" ref="userPreferences"/>

</bean>

In the preceding example, the singleton bean userManager is injected with a reference to the HTTP Session-scoped bean userPreferences. The salient point here is that the userManager bean is a singleton: it will be instantiated exactly once per container, and its dependencies (in this case only one, the userPreferences bean) are also injected only once. This means that the userManager bean will only operate on the exact same userPreferences object, that is, the one that it was originally injected with.

This is not the behavior you want when injecting a shorter-lived scoped bean into a longer-lived scoped bean, for example injecting an HTTP Session-scoped collaborating bean as a dependency into singleton bean. Rather, you need a single userManager object, and for the lifetime of an HTTP Session, you need a userPreferences object that is specific to said HTTP Session. Thus the container creates an object that exposes the exact same public interface as the UserPreferences class (ideally an object that is a UserPreferences instance) which can fetch the real UserPreferences object from the scoping mechanism (HTTP request, Session, etc.). The container injects this proxy object into the userManager bean, which is unaware that this UserPreferences reference is a proxy. In this example, when a UserManager instance invokes a method on the dependency-injected UserPreferences object, it actually is invoking a method on the proxy. The proxy then fetches the real UserPreferences object from (in this case) the HTTP Session, and delegates the method invocation onto the retrieved real UserPreferences object.

Thus you need the following, correct and complete, configuration when injecting request- and session-scoped beans into collaborating objects:

<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

<aop:scoped-proxy/>

</bean>

<bean id="userManager" class="com.foo.UserManager">

<property name="userPreferences" ref="userPreferences"/>

</bean>

Choosing the type of proxy to create

By default, when the Spring container creates a proxy for a bean that is marked up with the <aop:scoped-proxy/> element, a CGLIB-based class proxy is created.

CGLIB proxies only intercept public method calls! Do not call non-public methods on such a proxy; they will not be delegated to the actual scoped target object.

Alternatively, you can configure the Spring container to create standard JDK interface-based proxies for such scoped beans, by specifying false for the value of the proxy-target-class attribute of the <aop:scoped-proxy/> element. Using JDK interface-based proxies means that you do not need additional libraries in your application classpath to effect such proxying. However, it also means that the class of the scoped bean must implement at least one interface, and that all collaborators into which the scoped bean is injected must reference the bean through one of its interfaces.

<!-- DefaultUserPreferences implements the UserPreferences interface -->

<bean id="userPreferences" class="com.foo.DefaultUserPreferences" scope="session">

<aop:scoped-proxy proxy-target-class="false"/>

</bean>

<bean id="userManager" class="com.foo.UserManager">

<property name="userPreferences" ref="userPreferences"/>

</bean>

For more detailed information about choosing class-based or interface-based proxying, see Proxying mechanisms.

1.5.5. Custom scopes

The bean scoping mechanism is extensible; You can define your own scopes, or even redefine existing scopes, although the latter is considered bad practice and you cannot override the built-in singleton and prototype scopes.

Creating a custom scope

To integrate your custom scope(s) into the Spring container, you need to implement the org.springframework.beans.factory.config.Scope interface, which is described in this section. For an idea of how to implement your own scopes, see the Scope implementations that are supplied with the Spring Framework itself and the Scope javadocs, which explains the methods you need to implement in more detail.

The Scope interface has four methods to get objects from the scope, remove them from the scope, and allow them to be destroyed.

The following method returns the object from the underlying scope. The session scope implementation, for example, returns the session-scoped bean (and if it does not exist, the method returns a new instance of the bean, after having bound it to the session for future reference).

Object get(String name, ObjectFactory objectFactory)

The following method removes the object from the underlying scope. The session scope implementation for example, removes the session-scoped bean from the underlying session. The object should be returned, but you can return null if the object with the specified name is not found.

Object remove(String name)

The following method registers the callbacks the scope should execute when it is destroyed or when the specified object in the scope is destroyed. Refer to the javadocs or a Spring scope implementation for more information on destruction callbacks.

void registerDestructionCallback(String name, Runnable destructionCallback)

The following method obtains the conversation identifier for the underlying scope. This identifier is different for each scope. For a session scoped implementation, this identifier can be the session identifier.

String getConversationId()

Using a custom scope

After you write and test one or more custom Scope implementations, you need to make the Spring container aware of your new scope(s). The following method is the central method to register a new Scope with the Spring container:

void registerScope(String scopeName, Scope scope);

This method is declared on the ConfigurableBeanFactory interface, which is available on most of the concrete ApplicationContext implementations that ship with Spring via the BeanFactory property.

The first argument to the registerScope(..) method is the unique name associated with a scope; examples of such names in the Spring container itself are singleton and prototype. The second argument to the registerScope(..) method is an actual instance of the custom Scope implementation that you wish to register and use.

Suppose that you write your custom Scope implementation, and then register it as below.

The example below uses SimpleThreadScope which is included with Spring, but not registered by default. The instructions would be the same for your own custom Scope implementations.

Scope threadScope = new SimpleThreadScope();

beanFactory.registerScope("thread", threadScope);

You then create bean definitions that adhere to the scoping rules of your custom Scope:

<bean id="..." class="..." scope="thread">

With a custom Scope implementation, you are not limited to programmatic registration of the scope. You can also do the Scope registration declaratively, using the CustomScopeConfigurer class:

<?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:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">

<property name="scopes">

<map>

<entry key="thread">

<bean class="org.springframework.context.support.SimpleThreadScope"/>

</entry>

</map>

</property>

</bean>

<bean id="bar" class="x.y.Bar" scope="thread">

<property name="name" value="Rick"/>

<aop:scoped-proxy/>

</bean>

<bean id="foo" class="x.y.Foo">

<property name="bar" ref="bar"/>

</bean>

</beans>

When you place <aop:scoped-proxy/> in a FactoryBean implementation, it is the factory bean itself that is scoped, not the object returned from getObject().

1.6. Customizing the nature of a bean

1.6.1. Lifecycle callbacks

To interact with the container’s management of the bean lifecycle, you can implement the Spring InitializingBean and DisposableBean interfaces. The container calls afterPropertiesSet() for the former and destroy() for the latter to allow the bean to perform certain actions upon initialization and destruction of your beans.

The JSR-250 @PostConstruct and @PreDestroy annotations are generally considered best practice for receiving lifecycle callbacks in a modern Spring application. Using these annotations means that your beans are not coupled to Spring specific interfaces. For details see @PostConstruct and @PreDestroy.

If you don’t want to use the JSR-250 annotations but you are still looking to remove coupling consider the use of init-method and destroy-method object definition metadata.

Internally, the Spring Framework uses BeanPostProcessor implementations to process any callback interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle behavior Spring does not offer out-of-the-box, you can implement a BeanPostProcessor yourself. For more information, see Container Extension Points.

In addition to the initialization and destruction callbacks, Spring-managed objects may also implement the Lifecycle interface so that those objects can participate in the startup and shutdown process as driven by the container’s own lifecycle.

The lifecycle callback interfaces are described in this section.

Initialization callbacks

The org.springframework.beans.factory.InitializingBean interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container. The InitializingBean interface specifies a single method:

void afterPropertiesSet() throws Exception;

It is recommended that you do not use the InitializingBean interface because it unnecessarily couples the code to Spring. Alternatively, use the @PostConstruct annotation or specify a POJO initialization method. In the case of XML-based configuration metadata, you use the init-method attribute to specify the name of the method that has a void no-argument signature. With Java config, you use the initMethod attribute of @Bean, see Receiving lifecycle callbacks. For example, the following:

<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>

public class ExampleBean {

public void init() {

// do some initialization work

}

}

…​is exactly the same as…​

<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

public class AnotherExampleBean implements InitializingBean {

public void afterPropertiesSet() {

// do some initialization work

}

}

but does not couple the code to Spring.

Destruction callbacks

Implementing the org.springframework.beans.factory.DisposableBean interface allows a bean to get a callback when the container containing it is destroyed. The DisposableBean interface specifies a single method:

void destroy() throws Exception;

It is recommended that you do not use the DisposableBean callback interface because it unnecessarily couples the code to Spring. Alternatively, use the @PreDestroy annotation or specify a generic method that is supported by bean definitions. With XML-based configuration metadata, you use the destroy-method attribute on the <bean/>. With Java config, you use the destroyMethod attribute of @Bean, see Receiving lifecycle callbacks. For example, the following definition:

<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>

public class ExampleBean {

public void cleanup() {

// do some destruction work (like releasing pooled connections)

}

}

is exactly the same as:

<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

public class AnotherExampleBean implements DisposableBean {

public void destroy() {

// do some destruction work (like releasing pooled connections)

}

}

but does not couple the code to Spring.

The destroy-method attribute of a <bean> element can be assigned a special (inferred) value which instructs Spring to automatically detect a public close or shutdown method on the specific bean class (any class that implements java.lang.AutoCloseable or java.io.Closeable would therefore match). This special (inferred) value can also be set on the default-destroy-method attribute of a <beans> element to apply this behavior to an entire set of beans (see Default initialization and destroy methods). Note that this is the default behavior with Java config.

Default initialization and destroy methods

When you write initialization and destroy method callbacks that do not use the Spring-specific InitializingBean and DisposableBean callback interfaces, you typically write methods with names such as init(), initialize(), dispose(), and so on. Ideally, the names of such lifecycle callback methods are standardized across a project so that all developers use the same method names and ensure consistency.

You can configure the Spring container to look for named initialization and destroy callback method names on every bean. This means that you, as an application developer, can write your application classes and use an initialization callback called init(), without having to configure an init-method="init" attribute with each bean definition. The Spring IoC container calls that method when the bean is created (and in accordance with the standard lifecycle callback contract described previously). This feature also enforces a consistent naming convention for initialization and destroy method callbacks.

Suppose that your initialization callback methods are named init() and destroy callback methods are named destroy(). Your class will resemble the class in the following example.

public class DefaultBlogService implements BlogService {

private BlogDao blogDao;

public void setBlogDao(BlogDao blogDao) {

this.blogDao = blogDao;

}

// this is (unsurprisingly) the initialization callback method

public void init() {

if (this.blogDao == null) {

throw new IllegalStateException("The [blogDao] property must be set.");

}

}

}

<beans default-init-method="init">

<bean id="blogService" class="com.foo.DefaultBlogService">

<property name="blogDao" ref="blogDao" />

</bean>

</beans>

The presence of the default-init-method attribute on the top-level <beans/> element attribute causes the Spring IoC container to recognize a method called init on beans as the initialization method callback. When a bean is created and assembled, if the bean class has such a method, it is invoked at the appropriate time.

You configure destroy method callbacks similarly (in XML, that is) by using the default-destroy-method attribute on the top-level <beans/> element.

Where existing bean classes already have callback methods that are named at variance with the convention, you can override the default by specifying (in XML, that is) the method name using the init-method and destroy-method attributes of the <bean/> itself.

The Spring container guarantees that a configured initialization callback is called immediately after a bean is supplied with all dependencies. Thus the initialization callback is called on the raw bean reference, which means that AOP interceptors and so forth are not yet applied to the bean. A target bean is fully created first, then an AOP proxy (for example) with its interceptor chain is applied. If the target bean and the proxy are defined separately, your code can even interact with the raw target bean, bypassing the proxy. Hence, it would be inconsistent to apply the interceptors to the init method, because doing so would couple the lifecycle of the target bean with its proxy/interceptors and leave strange semantics when your code interacts directly to the raw target bean.

Combining lifecycle mechanisms

As of Spring 2.5, you have three options for controlling bean lifecycle behavior: the InitializingBean and DisposableBean callback interfaces; custom init() and destroy() methods; and the @PostConstruct and @PreDestroy annotations. You can combine these mechanisms to control a given bean.

If multiple lifecycle mechanisms are configured for a bean, and each mechanism is configured with a different method name, then each configured method is executed in the order listed below. However, if the same method name is configured - for example, init() for an initialization method - for more than one of these lifecycle mechanisms, that method is executed once, as explained in the preceding section.

Multiple lifecycle mechanisms configured for the same bean, with different initialization methods, are called as follows:

Methods annotated with @PostConstruct

afterPropertiesSet() as defined by the InitializingBean callback interface

A custom configured init() method

Destroy methods are called in the same order:

Methods annotated with @PreDestroy

destroy() as defined by the DisposableBean callback interface

A custom configured destroy() method

Startup and shutdown callbacks

The Lifecycle interface defines the essential methods for any object that has its own lifecycle requirements (e.g. starts and stops some background process):

public interface Lifecycle {

void start();

void stop();

boolean isRunning();

}

Any Spring-managed object may implement that interface. Then, when the ApplicationContext itself receives start and stop signals, e.g. for a stop/restart scenario at runtime, it will cascade those calls to all Lifecycle implementations defined within that context. It does this by delegating to a LifecycleProcessor:

public interface LifecycleProcessor extends Lifecycle {

void onRefresh();

void onClose();

}

Notice that the LifecycleProcessor is itself an extension of the Lifecycle interface. It also adds two other methods for reacting to the context being refreshed and closed.

Note that the regular org.springframework.context.Lifecycle interface is just a plain contract for explicit start/stop notifications and does NOT imply auto-startup at context refresh time. Consider implementing org.springframework.context.SmartLifecycle instead for fine-grained control over auto-startup of a specific bean (including startup phases). Also, please note that stop notifications are not guaranteed to come before destruction: On regular shutdown, all Lifecycle beans will first receive a stop notification before the general destruction callbacks are being propagated; however, on hot refresh during a context’s lifetime or on aborted refresh attempts, only destroy methods will be called.

The order of startup and shutdown invocations can be important. If a "depends-on" relationship exists between any two objects, the dependent side will start after its dependency, and it will stop before its dependency. However, at times the direct dependencies are unknown. You may only know that objects of a certain type should start prior to objects of another type. In those cases, the SmartLifecycle interface defines another option, namely the getPhase() method as defined on its super-interface, Phased.

public interface Phased {

int getPhase();

}

public interface SmartLifecycle extends Lifecycle, Phased {

boolean isAutoStartup();

void stop(Runnable callback);

}

When starting, the objects with the lowest phase start first, and when stopping, the reverse order is followed. Therefore, an object that implements SmartLifecycle and whose getPhase() method returns Integer.MIN\_VALUE would be among the first to start and the last to stop. At the other end of the spectrum, a phase value of Integer.MAX\_VALUE would indicate that the object should be started last and stopped first (likely because it depends on other processes to be running). When considering the phase value, it’s also important to know that the default phase for any "normal" Lifecycle object that does not implement SmartLifecycle would be 0. Therefore, any negative phase value would indicate that an object should start before those standard components (and stop after them), and vice versa for any positive phase value.

As you can see the stop method defined by SmartLifecycle accepts a callback. Any implementation must invoke that callback’s run() method after that implementation’s shutdown process is complete. That enables asynchronous shutdown where necessary since the default implementation of the LifecycleProcessor interface, DefaultLifecycleProcessor, will wait up to its timeout value for the group of objects within each phase to invoke that callback. The default per-phase timeout is 30 seconds. You can override the default lifecycle processor instance by defining a bean named "lifecycleProcessor" within the context. If you only want to modify the timeout, then defining the following would be sufficient:

<bean id="lifecycleProcessor" class="org.springframework.context.support.DefaultLifecycleProcessor">

<!-- timeout value in milliseconds -->

<property name="timeoutPerShutdownPhase" value="10000"/>

</bean>

As mentioned, the LifecycleProcessor interface defines callback methods for the refreshing and closing of the context as well. The latter will simply drive the shutdown process as if stop() had been called explicitly, but it will happen when the context is closing. The 'refresh' callback on the other hand enables another feature of SmartLifecycle beans. When the context is refreshed (after all objects have been instantiated and initialized), that callback will be invoked, and at that point the default lifecycle processor will check the boolean value returned by each SmartLifecycle object’s isAutoStartup() method. If "true", then that object will be started at that point rather than waiting for an explicit invocation of the context’s or its own start() method (unlike the context refresh, the context start does not happen automatically for a standard context implementation). The "phase" value as well as any "depends-on" relationships will determine the startup order in the same way as described above.

Shutting down the Spring IoC container gracefully in non-web applications

This section applies only to non-web applications. Spring’s web-based ApplicationContext implementations already have code in place to shut down the Spring IoC container gracefully when the relevant web application is shut down.

If you are using Spring’s IoC container in a non-web application environment; for example, in a rich client desktop environment; you register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released. Of course, you must still configure and implement these destroy callbacks correctly.

To register a shutdown hook, you call the registerShutdownHook() method that is declared on the ConfigurableApplicationContext interface:

import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Boot {

public static void main(final String[] args) throws Exception {

ConfigurableApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");

// add a shutdown hook for the above context...

ctx.registerShutdownHook();

// app runs here...

// main method exits, hook is called prior to the app shutting down...

}

}

1.6.2. ApplicationContextAware and BeanNameAware

When an ApplicationContext creates an object instance that implements the org.springframework.context.ApplicationContextAware interface, the instance is provided with a reference to that ApplicationContext.

public interface ApplicationContextAware {

void setApplicationContext(ApplicationContext applicationContext) throws BeansException;

}

Thus beans can manipulate programmatically the ApplicationContext that created them, through the ApplicationContext interface, or by casting the reference to a known subclass of this interface, such as ConfigurableApplicationContext, which exposes additional functionality. One use would be the programmatic retrieval of other beans. Sometimes this capability is useful; however, in general you should avoid it, because it couples the code to Spring and does not follow the Inversion of Control style, where collaborators are provided to beans as properties. Other methods of the ApplicationContext provide access to file resources, publishing application events, and accessing a MessageSource. These additional features are described in Additional capabilities of the ApplicationContext

As of Spring 2.5, autowiring is another alternative to obtain reference to the ApplicationContext. The "traditional" constructor and byType autowiring modes (as described in Autowiring collaborators) can provide a dependency of type ApplicationContext for a constructor argument or setter method parameter, respectively. For more flexibility, including the ability to autowire fields and multiple parameter methods, use the new annotation-based autowiring features. If you do, the ApplicationContext is autowired into a field, constructor argument, or method parameter that is expecting the ApplicationContext type if the field, constructor, or method in question carries the @Autowired annotation. For more information, see @Autowired.

When an ApplicationContext creates a class that implements the org.springframework.beans.factory.BeanNameAware interface, the class is provided with a reference to the name defined in its associated object definition.

public interface BeanNameAware {

void setBeanName(String name) throws BeansException;

}

The callback is invoked after population of normal bean properties but before an initialization callback such as InitializingBean afterPropertiesSet or a custom init-method.

1.6.3. Other Aware interfaces

Besides ApplicationContextAware and BeanNameAware discussed above, Spring offers a range of Aware interfaces that allow beans to indicate to the container that they require a certain infrastructure dependency. The most important Aware interfaces are summarized below - as a general rule, the name is a good indication of the dependency type:

Table 4. Aware interfaces

Name Injected Dependency Explained in…​

ApplicationContextAware

Declaring ApplicationContext

ApplicationContextAware and BeanNameAware

ApplicationEventPublisherAware

Event publisher of the enclosing ApplicationContext

Additional capabilities of the ApplicationContext

BeanClassLoaderAware

Class loader used to load the bean classes.

Instantiating beans

BeanFactoryAware

Declaring BeanFactory

ApplicationContextAware and BeanNameAware

BeanNameAware

Name of the declaring bean

ApplicationContextAware and BeanNameAware

BootstrapContextAware

Resource adapter BootstrapContext the container runs in. Typically available only in JCA aware ApplicationContexts

JCA CCI

LoadTimeWeaverAware

Defined weaver for processing class definition at load time

Load-time weaving with AspectJ in the Spring Framework

MessageSourceAware

Configured strategy for resolving messages (with support for parametrization and internationalization)

Additional capabilities of the ApplicationContext

NotificationPublisherAware

Spring JMX notification publisher

Notifications

ResourceLoaderAware

Configured loader for low-level access to resources

Resources

ServletConfigAware

Current ServletConfig the container runs in. Valid only in a web-aware Spring ApplicationContext

Spring MVC

ServletContextAware

Current ServletContext the container runs in. Valid only in a web-aware Spring ApplicationContext

Spring MVC

Note again that usage of these interfaces ties your code to the Spring API and does not follow the Inversion of Control style. As such, they are recommended for infrastructure beans that require programmatic access to the container.

1.7. Bean definition inheritance

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on. A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed. Using parent and child bean definitions can save a lot of typing. Effectively, this is a form of templating.

If you work with an ApplicationContext interface programmatically, child bean definitions are represented by the ChildBeanDefinition class. Most users do not work with them on this level, instead configuring bean definitions declaratively in something like the ClassPathXmlApplicationContext. When you use XML-based configuration metadata, you indicate a child bean definition by using the parent attribute, specifying the parent bean as the value of this attribute.

<bean id="inheritedTestBean" abstract="true"

class="org.springframework.beans.TestBean">

<property name="name" value="parent"/>

<property name="age" value="1"/>

</bean>

<bean id="inheritsWithDifferentClass"

class="org.springframework.beans.DerivedTestBean"

parent="inheritedTestBean" init-method="initialize">

<property name="name" value="override"/>

<!-- the age property value of 1 will be inherited from parent -->

</bean>

A child bean definition uses the bean class from the parent definition if none is specified, but can also override it. In the latter case, the child bean class must be compatible with the parent, that is, it must accept the parent’s property values.

A child bean definition inherits scope, constructor argument values, property values, and method overrides from the parent, with the option to add new values. Any scope, initialization method, destroy method, and/or static factory method settings that you specify will override the corresponding parent settings.

The remaining settings are always taken from the child definition: depends on, autowire mode, dependency check, singleton, lazy init.

The preceding example explicitly marks the parent bean definition as abstract by using the abstract attribute. If the parent definition does not specify a class, explicitly marking the parent bean definition as abstract is required, as follows:

<bean id="inheritedTestBeanWithoutClass" abstract="true">

<property name="name" value="parent"/>

<property name="age" value="1"/>

</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"

parent="inheritedTestBeanWithoutClass" init-method="initialize">

<property name="name" value="override"/>

<!-- age will inherit the value of 1 from the parent bean definition-->

</bean>

The parent bean cannot be instantiated on its own because it is incomplete, and it is also explicitly marked as abstract. When a definition is abstract like this, it is usable only as a pure template bean definition that serves as a parent definition for child definitions. Trying to use such an abstract parent bean on its own, by referring to it as a ref property of another bean or doing an explicit getBean() call with the parent bean id, returns an error. Similarly, the container’s internal preInstantiateSingletons() method ignores bean definitions that are defined as abstract.

ApplicationContext pre-instantiates all singletons by default. Therefore, it is important (at least for singleton beans) that if you have a (parent) bean definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the abstract attribute to true, otherwise the application context will actually (attempt to) pre-instantiate the abstract bean.

1.8. Container Extension Points

Typically, an application developer does not need to subclass ApplicationContext implementation classes. Instead, the Spring IoC container can be extended by plugging in implementations of special integration interfaces. The next few sections describe these integration interfaces.

1.8.1. Customizing beans using a BeanPostProcessor

The BeanPostProcessor interface defines callback methods that you can implement to provide your own (or override the container’s default) instantiation logic, dependency-resolution logic, and so forth. If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more BeanPostProcessor implementations.

You can configure multiple BeanPostProcessor instances, and you can control the order in which these BeanPostProcessors execute by setting the order property. You can set this property only if the BeanPostProcessor implements the Ordered interface; if you write your own BeanPostProcessor you should consider implementing the Ordered interface too. For further details, consult the javadocs of the BeanPostProcessor and Ordered interfaces. See also the note below on programmatic registration of BeanPostProcessors.

BeanPostProcessors operate on bean (or object) instances; that is to say, the Spring IoC container instantiates a bean instance and then BeanPostProcessors do their work.

BeanPostProcessors are scoped per-container. This is only relevant if you are using container hierarchies. If you define a BeanPostProcessor in one container, it will only post-process the beans in that container. In other words, beans that are defined in one container are not post-processed by a BeanPostProcessor defined in another container, even if both containers are part of the same hierarchy.

To change the actual bean definition (i.e., the blueprint that defines the bean), you instead need to use a BeanFactoryPostProcessor as described in Customizing configuration metadata with a BeanFactoryPostProcessor.

The org.springframework.beans.factory.config.BeanPostProcessor interface consists of exactly two callback methods. When such a class is registered as a post-processor with the container, for each bean instance that is created by the container, the post-processor gets a callback from the container both before container initialization methods (such as InitializingBean’s afterPropertiesSet() and any declared init method) are called as well as after any bean initialization callbacks. The post-processor can take any action with the bean instance, including ignoring the callback completely. A bean post-processor typically checks for callback interfaces or may wrap a bean with a proxy. Some Spring AOP infrastructure classes are implemented as bean post-processors in order to provide proxy-wrapping logic.

An ApplicationContext automatically detects any beans that are defined in the configuration metadata which implement the BeanPostProcessor interface. The ApplicationContext registers these beans as post-processors so that they can be called later upon bean creation. Bean post-processors can be deployed in the container just like any other beans.

Note that when declaring a BeanPostProcessor using an @Bean factory method on a configuration class, the return type of the factory method should be the implementation class itself or at least the org.springframework.beans.factory.config.BeanPostProcessor interface, clearly indicating the post-processor nature of that bean. Otherwise, the ApplicationContext won’t be able to autodetect it by type before fully creating it. Since a BeanPostProcessor needs to be instantiated early in order to apply to the initialization of other beans in the context, this early type detection is critical.

Programmatically registering BeanPostProcessors

While the recommended approach for BeanPostProcessor registration is through ApplicationContext auto-detection (as described above), it is also possible to register them programmatically against a ConfigurableBeanFactory using the addBeanPostProcessor method. This can be useful when needing to evaluate conditional logic before registration, or even for copying bean post processors across contexts in a hierarchy. Note however that BeanPostProcessors added programmatically do not respect the Ordered interface. Here it is the order of registration that dictates the order of execution. Note also that BeanPostProcessors registered programmatically are always processed before those registered through auto-detection, regardless of any explicit ordering.

BeanPostProcessors and AOP auto-proxying

Classes that implement the BeanPostProcessor interface are special and are treated differently by the container. All BeanPostProcessors and beans that they reference directly are instantiated on startup, as part of the special startup phase of the ApplicationContext. Next, all BeanPostProcessors are registered in a sorted fashion and applied to all further beans in the container. Because AOP auto-proxying is implemented as a BeanPostProcessor itself, neither BeanPostProcessors nor the beans they reference directly are eligible for auto-proxying, and thus do not have aspects woven into them.

For any such bean, you should see an informational log message: "Bean foo is not eligible for getting processed by all BeanPostProcessor interfaces (for example: not eligible for auto-proxying)".

Note that if you have beans wired into your BeanPostProcessor using autowiring or @Resource (which may fall back to autowiring), Spring might access unexpected beans when searching for type-matching dependency candidates, and therefore make them ineligible for auto-proxying or other kinds of bean post-processing. For example, if you have a dependency annotated with @Resource where the field/setter name does not directly correspond to the declared name of a bean and no name attribute is used, then Spring will access other beans for matching them by type.

The following examples show how to write, register, and use BeanPostProcessors in an ApplicationContext.

Example: Hello World, BeanPostProcessor-style

This first example illustrates basic usage. The example shows a custom BeanPostProcessor implementation that invokes the toString() method of each bean as it is created by the container and prints the resulting string to the system console.

Find below the custom BeanPostProcessor implementation class definition:

package scripting;

import org.springframework.beans.factory.config.BeanPostProcessor;

public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {

// simply return the instantiated bean as-is

public Object postProcessBeforeInitialization(Object bean, String beanName) {

return bean; // we could potentially return any object reference here...

}

public Object postProcessAfterInitialization(Object bean, String beanName) {

System.out.println("Bean '" + beanName + "' created : " + bean.toString());

return bean;

}

}

<?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:lang="http://www.springframework.org/schema/lang"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/lang

http://www.springframework.org/schema/lang/spring-lang.xsd">

<lang:groovy id="messenger"

script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">

<lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>

</lang:groovy>

<!--

when the above bean (messenger) is instantiated, this custom

BeanPostProcessor implementation will output the fact to the system console

-->

<bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>

Notice how the InstantiationTracingBeanPostProcessor is simply defined. It does not even have a name, and because it is a bean it can be dependency-injected just like any other bean. (The preceding configuration also defines a bean that is backed by a Groovy script. The Spring dynamic language support is detailed in the chapter entitled Dynamic language support.)

The following simple Java application executes the preceding code and configuration:

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.springframework.scripting.Messenger;

public final class Boot {

public static void main(final String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("scripting/beans.xml");

Messenger messenger = (Messenger) ctx.getBean("messenger");

System.out.println(messenger);

}

}

The output of the preceding application resembles the following:

Bean 'messenger' created : org.springframework.scripting.groovy.GroovyMessenger@272961

org.springframework.scripting.groovy.GroovyMessenger@272961

Example: The RequiredAnnotationBeanPostProcessor

Using callback interfaces or annotations in conjunction with a custom BeanPostProcessor implementation is a common means of extending the Spring IoC container. An example is Spring’s RequiredAnnotationBeanPostProcessor - a BeanPostProcessor implementation that ships with the Spring distribution which ensures that JavaBean properties on beans that are marked with an (arbitrary) annotation are actually (configured to be) dependency-injected with a value.

1.8.2. Customizing configuration metadata with a BeanFactoryPostProcessor

The next extension point that we will look at is the org.springframework.beans.factory.config.BeanFactoryPostProcessor. The semantics of this interface are similar to those of the BeanPostProcessor, with one major difference: BeanFactoryPostProcessor operates on the bean configuration metadata; that is, the Spring IoC container allows a BeanFactoryPostProcessor to read the configuration metadata and potentially change it before the container instantiates any beans other than BeanFactoryPostProcessors.

You can configure multiple BeanFactoryPostProcessors, and you can control the order in which these BeanFactoryPostProcessors execute by setting the order property. However, you can only set this property if the BeanFactoryPostProcessor implements the Ordered interface. If you write your own BeanFactoryPostProcessor, you should consider implementing the Ordered interface too. Consult the javadocs of the BeanFactoryPostProcessor and Ordered interfaces for more details.

If you want to change the actual bean instances (i.e., the objects that are created from the configuration metadata), then you instead need to use a BeanPostProcessor (described above in Customizing beans using a BeanPostProcessor). While it is technically possible to work with bean instances within a BeanFactoryPostProcessor (e.g., using BeanFactory.getBean()), doing so causes premature bean instantiation, violating the standard container lifecycle. This may cause negative side effects such as bypassing bean post processing.

Also, BeanFactoryPostProcessors are scoped per-container. This is only relevant if you are using container hierarchies. If you define a BeanFactoryPostProcessor in one container, it will only be applied to the bean definitions in that container. Bean definitions in one container will not be post-processed by BeanFactoryPostProcessors in another container, even if both containers are part of the same hierarchy.

A bean factory post-processor is executed automatically when it is declared inside an ApplicationContext, in order to apply changes to the configuration metadata that define the container. Spring includes a number of predefined bean factory post-processors, such as PropertyOverrideConfigurer and PropertyPlaceholderConfigurer. A custom BeanFactoryPostProcessor can also be used, for example, to register custom property editors.

An ApplicationContext automatically detects any beans that are deployed into it that implement the BeanFactoryPostProcessor interface. It uses these beans as bean factory post-processors, at the appropriate time. You can deploy these post-processor beans as you would any other bean.

As with BeanPostProcessors , you typically do not want to configure BeanFactoryPostProcessors for lazy initialization. If no other bean references a Bean(Factory)PostProcessor, that post-processor will not get instantiated at all. Thus, marking it for lazy initialization will be ignored, and the Bean(Factory)PostProcessor will be instantiated eagerly even if you set the default-lazy-init attribute to true on the declaration of your <beans /> element.

Example: the Class name substitution PropertyPlaceholderConfigurer

You use the PropertyPlaceholderConfigurer to externalize property values from a bean definition in a separate file using the standard Java Properties format. Doing so enables the person deploying an application to customize environment-specific properties such as database URLs and passwords, without the complexity or risk of modifying the main XML definition file or files for the container.

Consider the following XML-based configuration metadata fragment, where a DataSource with placeholder values is defined. The example shows properties configured from an external Properties file. At runtime, a PropertyPlaceholderConfigurer is applied to the metadata that will replace some properties of the DataSource. The values to replace are specified as placeholders of the form ${property-name} which follows the Ant / log4j / JSP EL style.

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations" value="classpath:com/foo/jdbc.properties"/>

</bean>

<bean id="dataSource" destroy-method="close"

class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="${jdbc.driverClassName}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

The actual values come from another file in the standard Java Properties format:

jdbc.driverClassName=org.hsqldb.jdbcDriver

jdbc.url=jdbc:hsqldb:hsql://production:9002

jdbc.username=sa

jdbc.password=root

Therefore, the string ${jdbc.username} is replaced at runtime with the value 'sa', and the same applies for other placeholder values that match keys in the properties file. The PropertyPlaceholderConfigurer checks for placeholders in most properties and attributes of a bean definition. Furthermore, the placeholder prefix and suffix can be customized.

With the context namespace introduced in Spring 2.5, it is possible to configure property placeholders with a dedicated configuration element. One or more locations can be provided as a comma-separated list in the location attribute.

<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>

The PropertyPlaceholderConfigurer not only looks for properties in the Properties file you specify. By default it also checks against the Java System properties if it cannot find a property in the specified properties files. You can customize this behavior by setting the systemPropertiesMode property of the configurer with one of the following three supported integer values:

never (0): Never check system properties

fallback (1): Check system properties if not resolvable in the specified properties files. This is the default.

override (2): Check system properties first, before trying the specified properties files. This allows system properties to override any other property source.

Consult the PropertyPlaceholderConfigurer javadocs for more information.

You can use the PropertyPlaceholderConfigurer to substitute class names, which is sometimes useful when you have to pick a particular implementation class at runtime. For example:

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<value>classpath:com/foo/strategy.properties</value>

</property>

<property name="properties">

<value>custom.strategy.class=com.foo.DefaultStrategy</value>

</property>

</bean>

<bean id="serviceStrategy" class="${custom.strategy.class}"/>

If the class cannot be resolved at runtime to a valid class, resolution of the bean fails when it is about to be created, which is during the preInstantiateSingletons() phase of an ApplicationContext for a non-lazy-init bean.

Example: the PropertyOverrideConfigurer

The PropertyOverrideConfigurer, another bean factory post-processor, resembles the PropertyPlaceholderConfigurer, but unlike the latter, the original definitions can have default values or no values at all for bean properties. If an overriding Properties file does not have an entry for a certain bean property, the default context definition is used.

Note that the bean definition is not aware of being overridden, so it is not immediately obvious from the XML definition file that the override configurer is being used. In case of multiple PropertyOverrideConfigurer instances that define different values for the same bean property, the last one wins, due to the overriding mechanism.

Properties file configuration lines take this format:

beanName.property=value

For example:

dataSource.driverClassName=com.mysql.jdbc.Driver

dataSource.url=jdbc:mysql:mydb

This example file can be used with a container definition that contains a bean called dataSource, which has driver and url properties.

Compound property names are also supported, as long as every component of the path except the final property being overridden is already non-null (presumably initialized by the constructors). In this example…​

foo.fred.bob.sammy=123

the sammy property of the bob property of the fred property of the foo bean is set to the scalar value 123.

Specified override values are always literal values; they are not translated into bean references. This convention also applies when the original value in the XML bean definition specifies a bean reference.

With the context namespace introduced in Spring 2.5, it is possible to configure property overriding with a dedicated configuration element:

<context:property-override location="classpath:override.properties"/>

1.8.3. Customizing instantiation logic with a FactoryBean

Implement the org.springframework.beans.factory.FactoryBean interface for objects that are themselves factories.

The FactoryBean interface is a point of pluggability into the Spring IoC container’s instantiation logic. If you have complex initialization code that is better expressed in Java as opposed to a (potentially) verbose amount of XML, you can create your own FactoryBean, write the complex initialization inside that class, and then plug your custom FactoryBean into the container.

The FactoryBean interface provides three methods:

Object getObject(): returns an instance of the object this factory creates. The instance can possibly be shared, depending on whether this factory returns singletons or prototypes.

boolean isSingleton(): returns true if this FactoryBean returns singletons, false otherwise.

Class getObjectType(): returns the object type returned by the getObject() method or null if the type is not known in advance.

The FactoryBean concept and interface is used in a number of places within the Spring Framework; more than 50 implementations of the FactoryBean interface ship with Spring itself.

When you need to ask a container for an actual FactoryBean instance itself instead of the bean it produces, preface the bean’s id with the ampersand symbol ( &) when calling the getBean() method of the ApplicationContext. So for a given FactoryBean with an id of myBean, invoking getBean("myBean") on the container returns the product of the FactoryBean; whereas, invoking getBean("&myBean") returns the FactoryBean instance itself.

1.9. Annotation-based container configuration

Are annotations better than XML for configuring Spring?

The introduction of annotation-based configurations raised the question of whether this approach is 'better' than XML. The short answer is it depends. The long answer is that each approach has its pros and cons, and usually it is up to the developer to decide which strategy suits them better. Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.

No matter the choice, Spring can accommodate both styles and even mix them together. It’s worth pointing out that through its JavaConfig option, Spring allows annotations to be used in a non-invasive way, without touching the target components source code and that in terms of tooling, all configuration styles are supported by the Spring Tool Suite.

An alternative to XML setups is provided by annotation-based configuration which rely on the bytecode metadata for wiring up components instead of angle-bracket declarations. Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration. As mentioned in Example: The RequiredAnnotationBeanPostProcessor, using a BeanPostProcessor in conjunction with annotations is a common means of extending the Spring IoC container. For example, Spring 2.0 introduced the possibility of enforcing required properties with the @Required annotation. Spring 2.5 made it possible to follow that same general approach to drive Spring’s dependency injection. Essentially, the @Autowired annotation provides the same capabilities as described in Autowiring collaborators but with more fine-grained control and wider applicability. Spring 2.5 also added support for JSR-250 annotations such as @PostConstruct, and @PreDestroy. Spring 3.0 added support for JSR-330 (Dependency Injection for Java) annotations contained in the javax.inject package such as @Inject and @Named. Details about those annotations can be found in the relevant section.

Annotation injection is performed before XML injection, thus the latter configuration will override the former for properties wired through both approaches.

As always, you can register them as individual bean definitions, but they can also be implicitly registered by including the following tag in an XML-based Spring configuration (notice the inclusion of the context namespace):

<?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:annotation-config/>

</beans>

(The implicitly registered post-processors include AutowiredAnnotationBeanPostProcessor, CommonAnnotationBeanPostProcessor, PersistenceAnnotationBeanPostProcessor, as well as the aforementioned RequiredAnnotationBeanPostProcessor.)

<context:annotation-config/> only looks for annotations on beans in the same application context in which it is defined. This means that, if you put <context:annotation-config/> in a WebApplicationContext for a DispatcherServlet, it only checks for @Autowired beans in your controllers, and not your services. See The DispatcherServlet for more information.

1.9.1. @Required

The @Required annotation applies to bean property setter methods, as in the following example:

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Required

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

// ...

}

This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring. The container throws an exception if the affected bean property has not been populated; this allows for eager and explicit failure, avoiding NullPointerExceptions or the like later on. It is still recommended that you put assertions into the bean class itself, for example, into an init method. Doing so enforces those required references and values even when you use the class outside of a container.

1.9.2. @Autowired

JSR 330’s @Inject annotation can be used in place of Spring’s @Autowired annotation in the examples below. See here for more details.

You can apply the @Autowired annotation to constructors:

public class MovieRecommender {

private final CustomerPreferenceDao customerPreferenceDao;

@Autowired

public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

// ...

}

As of Spring Framework 4.3, an @Autowired annotation on such a constructor is no longer necessary if the target bean only defines one constructor to begin with. However, if several constructors are available, at least one must be annotated to teach the container which one to use.

As expected, you can also apply the @Autowired annotation to "traditional" setter methods:

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Autowired

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

// ...

}

You can also apply the annotation to methods with arbitrary names and/or multiple arguments:

public class MovieRecommender {

private MovieCatalog movieCatalog;

private CustomerPreferenceDao customerPreferenceDao;

@Autowired

public void prepare(MovieCatalog movieCatalog,

CustomerPreferenceDao customerPreferenceDao) {

this.movieCatalog = movieCatalog;

this.customerPreferenceDao = customerPreferenceDao;

}

// ...

}

You can apply @Autowired to fields as well and even mix it with constructors:

public class MovieRecommender {

private final CustomerPreferenceDao customerPreferenceDao;

@Autowired

private MovieCatalog movieCatalog;

@Autowired

public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {

this.customerPreferenceDao = customerPreferenceDao;

}

// ...

}

Make sure that your target components (e.g. MovieCatalog, CustomerPreferenceDao) are consistently declared by the type that you are using for your @Autowired-annotated injection points. Otherwise injection may fail due to no type match found at runtime.

For XML-defined beans or component classes found through a classpath scan, the container usually knows the concrete type upfront. However, for @Bean factory methods, you need to make sure that the declared return type is sufficiently expressive. For components implementing several interfaces or for components potentially referred to by their implementation type, consider declaring the most specific return type on your factory method (at least as specific as required by the injection points referring to your bean).

It is also possible to provide all beans of a particular type from the ApplicationContext by adding the annotation to a field or method that expects an array of that type:

public class MovieRecommender {

@Autowired

private MovieCatalog[] movieCatalogs;

// ...

}

The same applies for typed collections:

public class MovieRecommender {

private Set<MovieCatalog> movieCatalogs;

@Autowired

public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {

this.movieCatalogs = movieCatalogs;

}

// ...

}

Your target beans can implement the org.springframework.core.Ordered interface or use the @Order or standard @Priority annotation if you want items in the array or list to be sorted in a specific order. Otherwise their order will follow the registration order of the corresponding target bean definitions in the container.

The @Order annotation may be declared at target class level but also on @Bean methods, potentially being very individual per bean definition (in case of multiple definitions with the same bean class). @Order values may influence priorities at injection points, but please be aware that they do not influence singleton startup order which is an orthogonal concern determined by dependency relationships and @DependsOn declarations.

Note that the standard javax.annotation.Priority annotation is not available at the @Bean level since it cannot be declared on methods. Its semantics can be modeled through @Order values in combination with @Primary on a single bean per type.

Even typed Maps can be autowired as long as the expected key type is String. The Map values will contain all beans of the expected type, and the keys will contain the corresponding bean names:

public class MovieRecommender {

private Map<String, MovieCatalog> movieCatalogs;

@Autowired

public void setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {

this.movieCatalogs = movieCatalogs;

}

// ...

}

By default, the autowiring fails whenever zero candidate beans are available; the default behavior is to treat annotated methods, constructors, and fields as indicating required dependencies. This behavior can be changed as demonstrated below.

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Autowired(required = false)

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

// ...

}

Only one annotated constructor per-class can be marked as required, but multiple non-required constructors can be annotated. In that case, each is considered among the candidates and Spring uses the greediest constructor whose dependencies can be satisfied, that is the constructor that has the largest number of arguments.

The required attribute of @Autowired is recommended over the @Required annotation. The required attribute indicates that the property is not required for autowiring purposes, the property is ignored if it cannot be autowired. @Required, on the other hand, is stronger in that it enforces the property that was set by any means supported by the container. If no value is injected, a corresponding exception is raised.

Alternatively, you may express the non-required nature of a particular dependency through Java 8’s java.util.Optional:

public class SimpleMovieLister {

@Autowired

public void setMovieFinder(Optional<MovieFinder> movieFinder) {

...

}

}

As of Spring Framework 5.0, you may also use an @Nullable annotation (of any kind in any package, e.g. javax.annotation.Nullable from JSR-305):

public class SimpleMovieLister {

@Autowired

public void setMovieFinder(@Nullable MovieFinder movieFinder) {

...

}

}

You can also use @Autowired for interfaces that are well-known resolvable dependencies: BeanFactory, ApplicationContext, Environment, ResourceLoader, ApplicationEventPublisher, and MessageSource. These interfaces and their extended interfaces, such as ConfigurableApplicationContext or ResourcePatternResolver, are automatically resolved, with no special setup necessary.

public class MovieRecommender {

@Autowired

private ApplicationContext context;

public MovieRecommender() {

}

// ...

}

@Autowired, @Inject, @Resource, and @Value annotations are handled by Spring BeanPostProcessor implementations which in turn means that you cannot apply these annotations within your own BeanPostProcessor or BeanFactoryPostProcessor types (if any). These types must be 'wired up' explicitly via XML or using a Spring @Bean method.

1.9.3. Fine-tuning annotation-based autowiring with @Primary

Because autowiring by type may lead to multiple candidates, it is often necessary to have more control over the selection process. One way to accomplish this is with Spring’s @Primary annotation. @Primary indicates that a particular bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency. If exactly one 'primary' bean exists among the candidates, it will be the autowired value.

Let’s assume we have the following configuration that defines firstMovieCatalog as the primary MovieCatalog.

@Configuration

public class MovieConfiguration {

@Bean

@Primary

public MovieCatalog firstMovieCatalog() { ... }

@Bean

public MovieCatalog secondMovieCatalog() { ... }

// ...

}

With such configuration, the following MovieRecommender will be autowired with the firstMovieCatalog.

public class MovieRecommender {

@Autowired

private MovieCatalog movieCatalog;

// ...

}

The corresponding bean definitions appear as follows.

<?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:annotation-config/>

<bean class="example.SimpleMovieCatalog" primary="true">

<!-- inject any dependencies required by this bean -->

</bean>

<bean class="example.SimpleMovieCatalog">

<!-- inject any dependencies required by this bean -->

</bean>

<bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>

1.9.4. Fine-tuning annotation-based autowiring with qualifiers

@Primary is an effective way to use autowiring by type with several instances when one primary candidate can be determined. When more control over the selection process is required, Spring’s @Qualifier annotation can be used. You can associate qualifier values with specific arguments, narrowing the set of type matches so that a specific bean is chosen for each argument. In the simplest case, this can be a plain descriptive value:

public class MovieRecommender {

@Autowired

@Qualifier("main")

private MovieCatalog movieCatalog;

// ...

}

The @Qualifier annotation can also be specified on individual constructor arguments or method parameters:

public class MovieRecommender {

private MovieCatalog movieCatalog;

private CustomerPreferenceDao customerPreferenceDao;

@Autowired

public void prepare(@Qualifier("main")MovieCatalog movieCatalog,

CustomerPreferenceDao customerPreferenceDao) {

this.movieCatalog = movieCatalog;

this.customerPreferenceDao = customerPreferenceDao;

}

// ...

}

The corresponding bean definitions appear as follows. The bean with qualifier value "main" is wired with the constructor argument that is qualified with the same value.

<?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:annotation-config/>

<bean class="example.SimpleMovieCatalog">

<qualifier value="main"/>

<!-- inject any dependencies required by this bean -->

</bean>

<bean class="example.SimpleMovieCatalog">

<qualifier value="action"/>

<!-- inject any dependencies required by this bean -->

</bean>

<bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>

For a fallback match, the bean name is considered a default qualifier value. Thus you can define the bean with an id "main" instead of the nested qualifier element, leading to the same matching result. However, although you can use this convention to refer to specific beans by name, @Autowired is fundamentally about type-driven injection with optional semantic qualifiers. This means that qualifier values, even with the bean name fallback, always have narrowing semantics within the set of type matches; they do not semantically express a reference to a unique bean id. Good qualifier values are "main" or "EMEA" or "persistent", expressing characteristics of a specific component that are independent from the bean id, which may be auto-generated in case of an anonymous bean definition like the one in the preceding example.

Qualifiers also apply to typed collections, as discussed above, for example, to Set<MovieCatalog>. In this case, all matching beans according to the declared qualifiers are injected as a collection. This implies that qualifiers do not have to be unique; they rather simply constitute filtering criteria. For example, you can define multiple MovieCatalog beans with the same qualifier value "action", all of which would be injected into a Set<MovieCatalog> annotated with @Qualifier("action").

Letting qualifier values select against target bean names, within the type-matching candidates, doesn’t even require a @Qualifier annotation at the injection point. If there is no other resolution indicator (e.g. a qualifier or a primary marker), for a non-unique dependency situation, Spring will match the injection point name (i.e. field name or parameter name) against the target bean names and choose the same-named candidate, if any.

That said, if you intend to express annotation-driven injection by name, do not primarily use @Autowired, even if is capable of selecting by bean name among type-matching candidates. Instead, use the JSR-250 @Resource annotation, which is semantically defined to identify a specific target component by its unique name, with the declared type being irrelevant for the matching process. @Autowired has rather different semantics: After selecting candidate beans by type, the specified String qualifier value will be considered within those type-selected candidates only, e.g. matching an "account" qualifier against beans marked with the same qualifier label.

For beans that are themselves defined as a collection/map or array type, @Resource is a fine solution, referring to the specific collection or array bean by unique name. That said, as of 4.3, collection/map and array types can be matched through Spring’s @Autowired type matching algorithm as well, as long as the element type information is preserved in @Bean return type signatures or collection inheritance hierarchies. In this case, qualifier values can be used to select among same-typed collections, as outlined in the previous paragraph.

As of 4.3, @Autowired also considers self references for injection, i.e. references back to the bean that is currently injected. Note that self injection is a fallback; regular dependencies on other components always have precedence. In that sense, self references do not participate in regular candidate selection and are therefore in particular never primary; on the contrary, they always end up as lowest precedence. In practice, use self references as a last resort only, e.g. for calling other methods on the same instance through the bean’s transactional proxy: Consider factoring out the affected methods to a separate delegate bean in such a scenario. Alternatively, use @Resource which may obtain a proxy back to the current bean by its unique name.

@Autowired applies to fields, constructors, and multi-argument methods, allowing for narrowing through qualifier annotations at the parameter level. By contrast, @Resource is supported only for fields and bean property setter methods with a single argument. As a consequence, stick with qualifiers if your injection target is a constructor or a multi-argument method.

You can create your own custom qualifier annotations. Simply define an annotation and provide the @Qualifier annotation within your definition:

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface Genre {

String value();

}

Then you can provide the custom qualifier on autowired fields and parameters:

public class MovieRecommender {

@Autowired

@Genre("Action")

private MovieCatalog actionCatalog;

private MovieCatalog comedyCatalog;

@Autowired

public void setComedyCatalog(@Genre("Comedy") MovieCatalog comedyCatalog) {

this.comedyCatalog = comedyCatalog;

}

// ...

}

Next, provide the information for the candidate bean definitions. You can add <qualifier/> tags as sub-elements of the <bean/> tag and then specify the type and value to match your custom qualifier annotations. The type is matched against the fully-qualified class name of the annotation. Or, as a convenience if no risk of conflicting names exists, you can use the short class name. Both approaches are demonstrated in the following example.

<?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:annotation-config/>

<bean class="example.SimpleMovieCatalog">

<qualifier type="Genre" value="Action"/>

<!-- inject any dependencies required by this bean -->

</bean>

<bean class="example.SimpleMovieCatalog">

<qualifier type="example.Genre" value="Comedy"/>

<!-- inject any dependencies required by this bean -->

</bean>

<bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>

In Classpath scanning and managed components, you will see an annotation-based alternative to providing the qualifier metadata in XML. Specifically, see Providing qualifier metadata with annotations.

In some cases, it may be sufficient to use an annotation without a value. This may be useful when the annotation serves a more generic purpose and can be applied across several different types of dependencies. For example, you may provide an offline catalog that would be searched when no Internet connection is available. First define the simple annotation:

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface Offline {

}

Then add the annotation to the field or property to be autowired:

public class MovieRecommender {

@Autowired

@Offline

private MovieCatalog offlineCatalog;

// ...

}

Now the bean definition only needs a qualifier type:

<bean class="example.SimpleMovieCatalog">

<qualifier type="Offline"/>

<!-- inject any dependencies required by this bean -->

</bean>

You can also define custom qualifier annotations that accept named attributes in addition to or instead of the simple value attribute. If multiple attribute values are then specified on a field or parameter to be autowired, a bean definition must match all such attribute values to be considered an autowire candidate. As an example, consider the following annotation definition:

@Target({ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface MovieQualifier {

String genre();

Format format();

}

In this case Format is an enum:

public enum Format {

VHS, DVD, BLURAY

}

The fields to be autowired are annotated with the custom qualifier and include values for both attributes: genre and format.

public class MovieRecommender {

@Autowired

@MovieQualifier(format=Format.VHS, genre="Action")

private MovieCatalog actionVhsCatalog;

@Autowired

@MovieQualifier(format=Format.VHS, genre="Comedy")

private MovieCatalog comedyVhsCatalog;

@Autowired

@MovieQualifier(format=Format.DVD, genre="Action")

private MovieCatalog actionDvdCatalog;

@Autowired

@MovieQualifier(format=Format.BLURAY, genre="Comedy")

private MovieCatalog comedyBluRayCatalog;

// ...

}

Finally, the bean definitions should contain matching qualifier values. This example also demonstrates that bean meta attributes may be used instead of the <qualifier/> sub-elements. If available, the <qualifier/> and its attributes take precedence, but the autowiring mechanism falls back on the values provided within the <meta/> tags if no such qualifier is present, as in the last two bean definitions in the following example.

<?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:annotation-config/>

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

<attribute key="format" value="VHS"/>

<attribute key="genre" value="Action"/>

</qualifier>

<!-- inject any dependencies required by this bean -->

</bean>

<bean class="example.SimpleMovieCatalog">

<qualifier type="MovieQualifier">

<attribute key="format" value="VHS"/>

<attribute key="genre" value="Comedy"/>

</qualifier>

<!-- inject any dependencies required by this bean -->

</bean>

<bean class="example.SimpleMovieCatalog">

<meta key="format" value="DVD"/>

<meta key="genre" value="Action"/>

<!-- inject any dependencies required by this bean -->

</bean>

<bean class="example.SimpleMovieCatalog">

<meta key="format" value="BLURAY"/>

<meta key="genre" value="Comedy"/>

<!-- inject any dependencies required by this bean -->

</bean>

</beans>

1.9.5. Using generics as autowiring qualifiers

In addition to the @Qualifier annotation, it is also possible to use Java generic types as an implicit form of qualification. For example, suppose you have the following configuration:

@Configuration

public class MyConfiguration {

@Bean

public StringStore stringStore() {

return new StringStore();

}

@Bean

public IntegerStore integerStore() {

return new IntegerStore();

}

}

Assuming that beans above implement a generic interface, i.e. Store<String> and Store<Integer>, you can @Autowire the Store interface and the generic will be used as a qualifier:

@Autowired

private Store<String> s1; // <String> qualifier, injects the stringStore bean

@Autowired

private Store<Integer> s2; // <Integer> qualifier, injects the integerStore bean

Generic qualifiers also apply when autowiring Lists, Maps and Arrays:

// Inject all Store beans as long as they have an <Integer> generic

// Store<String> beans will not appear in this list

@Autowired

private List<Store<Integer>> s;

1.9.6. CustomAutowireConfigurer

The CustomAutowireConfigurer is a BeanFactoryPostProcessor that enables you to register your own custom qualifier annotation types even if they are not annotated with Spring’s @Qualifier annotation.

<bean id="customAutowireConfigurer"

class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">

<property name="customQualifierTypes">

<set>

<value>example.CustomQualifier</value>

</set>

</property>

</bean>

The AutowireCandidateResolver determines autowire candidates by:

the autowire-candidate value of each bean definition

any default-autowire-candidates pattern(s) available on the <beans/> element

the presence of @Qualifier annotations and any custom annotations registered with the CustomAutowireConfigurer

When multiple beans qualify as autowire candidates, the determination of a "primary" is the following: if exactly one bean definition among the candidates has a primary attribute set to true, it will be selected.

1.9.7. @Resource

Spring also supports injection using the JSR-250 @Resource annotation on fields or bean property setter methods. This is a common pattern in Java EE 5 and 6, for example in JSF 1.2 managed beans or JAX-WS 2.0 endpoints. Spring supports this pattern for Spring-managed objects as well.

@Resource takes a name attribute, and by default Spring interprets that value as the bean name to be injected. In other words, it follows by-name semantics, as demonstrated in this example:

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Resource(name="myMovieFinder")

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

If no name is specified explicitly, the default name is derived from the field name or setter method. In case of a field, it takes the field name; in case of a setter method, it takes the bean property name. So the following example is going to have the bean with name "movieFinder" injected into its setter method:

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Resource

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

The name provided with the annotation is resolved as a bean name by the ApplicationContext of which the CommonAnnotationBeanPostProcessor is aware. The names can be resolved through JNDI if you configure Spring’s SimpleJndiBeanFactory explicitly. However, it is recommended that you rely on the default behavior and simply use Spring’s JNDI lookup capabilities to preserve the level of indirection.

In the exclusive case of @Resource usage with no explicit name specified, and similar to @Autowired, @Resource finds a primary type match instead of a specific named bean and resolves well-known resolvable dependencies: the BeanFactory, ApplicationContext, ResourceLoader, ApplicationEventPublisher, and MessageSource interfaces.

Thus in the following example, the customerPreferenceDao field first looks for a bean named customerPreferenceDao, then falls back to a primary type match for the type CustomerPreferenceDao. The "context" field is injected based on the known resolvable dependency type ApplicationContext.

public class MovieRecommender {

@Resource

private CustomerPreferenceDao customerPreferenceDao;

@Resource

private ApplicationContext context;

public MovieRecommender() {

}

// ...

}

1.9.8. @PostConstruct and @PreDestroy

The CommonAnnotationBeanPostProcessor not only recognizes the @Resource annotation but also the JSR-250 lifecycle annotations. Introduced in Spring 2.5, the support for these annotations offers yet another alternative to those described in initialization callbacks and destruction callbacks. Provided that the CommonAnnotationBeanPostProcessor is registered within the Spring ApplicationContext, a method carrying one of these annotations is invoked at the same point in the lifecycle as the corresponding Spring lifecycle interface method or explicitly declared callback method. In the example below, the cache will be pre-populated upon initialization and cleared upon destruction.

public class CachingMovieLister {

@PostConstruct

public void populateMovieCache() {

// populates the movie cache upon initialization...

}

@PreDestroy

public void clearMovieCache() {

// clears the movie cache upon destruction...

}

}

For details about the effects of combining various lifecycle mechanisms, see Combining lifecycle mechanisms.

1.10. Classpath scanning and managed components

Most examples in this chapter use XML to specify the configuration metadata that produces each BeanDefinition within the Spring container. The previous section (Annotation-based container configuration) demonstrates how to provide a lot of the configuration metadata through source-level annotations. Even in those examples, however, the "base" bean definitions are explicitly defined in the XML file, while the annotations only drive the dependency injection. This section describes an option for implicitly detecting the candidate components by scanning the classpath. Candidate components are classes that match against a filter criteria and have a corresponding bean definition registered with the container. This removes the need to use XML to perform bean registration; instead you can use annotations (for example @Component), AspectJ type expressions, or your own custom filter criteria to select which classes will have bean definitions registered with the container.

Starting with Spring 3.0, many features provided by the Spring JavaConfig project are part of the core Spring Framework. This allows you to define beans using Java rather than using the traditional XML files. Take a look at the @Configuration, @Bean, @Import, and @DependsOn annotations for examples of how to use these new features.

1.10.1. @Component and further stereotype annotations

The @Repository annotation is a marker for any class that fulfills the role or stereotype of a repository (also known as Data Access Object or DAO). Among the uses of this marker is the automatic translation of exceptions as described in Exception translation.

Spring provides further stereotype annotations: @Component, @Service, and @Controller. @Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively. Therefore, you can annotate your component classes with @Component, but by annotating them with @Repository, @Service, or @Controller instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. It is also possible that @Repository, @Service, and @Controller may carry additional semantics in future releases of the Spring Framework. Thus, if you are choosing between using @Component or @Service for your service layer, @Service is clearly the better choice. Similarly, as stated above, @Repository is already supported as a marker for automatic exception translation in your persistence layer.

1.10.2. Meta-annotations

Many of the annotations provided by Spring can be used as meta-annotations in your own code. A meta-annotation is simply an annotation that can be applied to another annotation. For example, the @Service annotation mentioned above is meta-annotated with @Component:

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Component // Spring will see this and treat @Service in the same way as @Component

public @interface Service {

// ....

}

Meta-annotations can also be combined to create composed annotations. For example, the @RestController annotation from Spring MVC is composed of @Controller and @ResponseBody.

In addition, composed annotations may optionally redeclare attributes from meta-annotations to allow user customization. This can be particularly useful when you want to only expose a subset of the meta-annotation’s attributes. For example, Spring’s @SessionScope annotation hardcodes the scope name to session but still allows customization of the proxyMode.

@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Scope(WebApplicationContext.SCOPE\_SESSION)

public @interface SessionScope {

/\*\*

\* Alias for {@link Scope#proxyMode}.

\* <p>Defaults to {@link ScopedProxyMode#TARGET\_CLASS}.

\*/

@AliasFor(annotation = Scope.class)

ScopedProxyMode proxyMode() default ScopedProxyMode.TARGET\_CLASS;

}

@SessionScope can then be used without declaring the proxyMode as follows:

@Service

@SessionScope

public class SessionScopedService {

// ...

}

Or with an overridden value for the proxyMode as follows:

@Service

@SessionScope(proxyMode = ScopedProxyMode.INTERFACES)

public class SessionScopedUserService implements UserService {

// ...

}

For further details, consult the Spring Annotation Programming Model wiki page.

1.10.3. Automatically detecting classes and registering bean definitions

Spring can automatically detect stereotyped classes and register corresponding BeanDefinitions with the ApplicationContext. For example, the following two classes are eligible for such autodetection:

@Service

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Autowired

public SimpleMovieLister(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

}

@Repository

public class JpaMovieFinder implements MovieFinder {

// implementation elided for clarity

}

To autodetect these classes and register the corresponding beans, you need to add @ComponentScan to your @Configuration class, where the basePackages attribute is a common parent package for the two classes. (Alternatively, you can specify a comma/semicolon/space-separated list that includes the parent package of each class.)

@Configuration

@ComponentScan(basePackages = "org.example")

public class AppConfig {

...

}

For concision, the above may have used the value attribute of the annotation, i.e. @ComponentScan("org.example")

The following is an alternative using 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="org.example"/>

</beans>

The use of <context:component-scan> implicitly enables the functionality of <context:annotation-config>. There is usually no need to include the <context:annotation-config> element when using <context:component-scan>.

The scanning of classpath packages requires the presence of corresponding directory entries in the classpath. When you build JARs with Ant, make sure that you do not activate the files-only switch of the JAR task. Also, classpath directories may not get exposed based on security policies in some environments, e.g. standalone apps on JDK 1.7.0\_45 and higher (which requires 'Trusted-Library' setup in your manifests; see http://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources).

On JDK 9’s module path (Jigsaw), Spring’s classpath scanning generally works as expected. However, please make sure that your component classes are exported in your module-info descriptors; if you expect Spring to invoke non-public members of your classes, make sure that they are 'opened' (i.e. using an opens declaration instead of an exports declaration in your module-info descriptor).

Furthermore, the AutowiredAnnotationBeanPostProcessor and CommonAnnotationBeanPostProcessor are both included implicitly when you use the component-scan element. That means that the two components are autodetected and wired together - all without any bean configuration metadata provided in XML.

You can disable the registration of AutowiredAnnotationBeanPostProcessor and CommonAnnotationBeanPostProcessor by including the annotation-config attribute with a value of false.

1.10.4. Using filters to customize scanning

By default, classes annotated with @Component, @Repository, @Service, @Controller, or a custom annotation that itself is annotated with @Component are the only detected candidate components. However, you can modify and extend this behavior simply by applying custom filters. Add them as includeFilters or excludeFilters parameters of the @ComponentScan annotation (or as include-filter or exclude-filter sub-elements of the component-scan element). Each filter element requires the type and expression attributes. The following table describes the filtering options.

Table 5. Filter Types

Filter Type Example Expression Description

annotation (default)

org.example.SomeAnnotation

An annotation to be present at the type level in target components.

assignable

org.example.SomeClass

A class (or interface) that the target components are assignable to (extend/implement).

aspectj

org.example..\*Service+

An AspectJ type expression to be matched by the target components.

regex

org\.example\.Default.\*

A regex expression to be matched by the target components class names.

custom

org.example.MyTypeFilter

A custom implementation of the org.springframework.core.type .TypeFilter interface.

The following example shows the configuration ignoring all @Repository annotations and using "stub" repositories instead.

@Configuration

@ComponentScan(basePackages = "org.example",

includeFilters = @Filter(type = FilterType.REGEX, pattern = ".\*Stub.\*Repository"),

excludeFilters = @Filter(Repository.class))

public class AppConfig {

...

}

and the equivalent using XML

<beans>

<context:component-scan base-package="org.example">

<context:include-filter type="regex"

expression=".\*Stub.\*Repository"/>

<context:exclude-filter type="annotation"

expression="org.springframework.stereotype.Repository"/>

</context:component-scan>

</beans>

You can also disable the default filters by setting useDefaultFilters=false on the annotation or providing use-default-filters="false" as an attribute of the <component-scan/> element. This will in effect disable automatic detection of classes annotated with @Component, @Repository, @Service, @Controller, or @Configuration.

1.10.5. Defining bean metadata within components

Spring components can also contribute bean definition metadata to the container. You do this with the same @Bean annotation used to define bean metadata within @Configuration annotated classes. Here is a simple example:

@Component

public class FactoryMethodComponent {

@Bean

@Qualifier("public")

public TestBean publicInstance() {

return new TestBean("publicInstance");

}

public void doWork() {

// Component method implementation omitted

}

}

This class is a Spring component that has application-specific code contained in its doWork() method. However, it also contributes a bean definition that has a factory method referring to the method publicInstance(). The @Bean annotation identifies the factory method and other bean definition properties, such as a qualifier value through the @Qualifier annotation. Other method level annotations that can be specified are @Scope, @Lazy, and custom qualifier annotations.

In addition to its role for component initialization, the @Lazy annotation may also be placed on injection points marked with @Autowired or @Inject. In this context, it leads to the injection of a lazy-resolution proxy.

Autowired fields and methods are supported as previously discussed, with additional support for autowiring of @Bean methods:

@Component

public class FactoryMethodComponent {

private static int i;

@Bean

@Qualifier("public")

public TestBean publicInstance() {

return new TestBean("publicInstance");

}

// use of a custom qualifier and autowiring of method parameters

@Bean

protected TestBean protectedInstance(

@Qualifier("public") TestBean spouse,

@Value("#{privateInstance.age}") String country) {

TestBean tb = new TestBean("protectedInstance", 1);

tb.setSpouse(spouse);

tb.setCountry(country);

return tb;

}

@Bean

private TestBean privateInstance() {

return new TestBean("privateInstance", i++);

}

@Bean

@RequestScope

public TestBean requestScopedInstance() {

return new TestBean("requestScopedInstance", 3);

}

}

The example autowires the String method parameter country to the value of the age property on another bean named privateInstance. A Spring Expression Language element defines the value of the property through the notation #{ <expression> }. For @Value annotations, an expression resolver is preconfigured to look for bean names when resolving expression text.

As of Spring Framework 4.3, you may also declare a factory method parameter of type InjectionPoint (or its more specific subclass DependencyDescriptor) in order to access the requesting injection point that triggers the creation of the current bean. Note that this will only apply to the actual creation of bean instances, not to the injection of existing instances. As a consequence, this feature makes most sense for beans of prototype scope. For other scopes, the factory method will only ever see the injection point which triggered the creation of a new bean instance in the given scope: for example, the dependency that triggered the creation of a lazy singleton bean. Use the provided injection point metadata with semantic care in such scenarios.

@Component

public class FactoryMethodComponent {

@Bean @Scope("prototype")

public TestBean prototypeInstance(InjectionPoint injectionPoint) {

return new TestBean("prototypeInstance for " + injectionPoint.getMember());

}

}

The @Bean methods in a regular Spring component are processed differently than their counterparts inside a Spring @Configuration class. The difference is that @Component classes are not enhanced with CGLIB to intercept the invocation of methods and fields. CGLIB proxying is the means by which invoking methods or fields within @Bean methods in @Configuration classes creates bean metadata references to collaborating objects; such methods are not invoked with normal Java semantics but rather go through the container in order to provide the usual lifecycle management and proxying of Spring beans even when referring to other beans via programmatic calls to @Bean methods. In contrast, invoking a method or field in an @Bean method within a plain @Component class has standard Java semantics, with no special CGLIB processing or other constraints applying.

You may declare @Bean methods as static, allowing for them to be called without creating their containing configuration class as an instance. This makes particular sense when defining post-processor beans, e.g. of type BeanFactoryPostProcessor or BeanPostProcessor, since such beans will get initialized early in the container lifecycle and should avoid triggering other parts of the configuration at that point.

Note that calls to static @Bean methods will never get intercepted by the container, not even within @Configuration classes (see above). This is due to technical limitations: CGLIB subclassing can only override non-static methods. As a consequence, a direct call to another @Bean method will have standard Java semantics, resulting in an independent instance being returned straight from the factory method itself.

The Java language visibility of @Bean methods does not have an immediate impact on the resulting bean definition in Spring’s container. You may freely declare your factory methods as you see fit in non-@Configuration classes and also for static methods anywhere. However, regular @Bean methods in @Configuration classes need to be overridable, i.e. they must not be declared as private or final.

@Bean methods will also be discovered on base classes of a given component or configuration class, as well as on Java 8 default methods declared in interfaces implemented by the component or configuration class. This allows for a lot of flexibility in composing complex configuration arrangements, with even multiple inheritance being possible through Java 8 default methods as of Spring 4.2.

Finally, note that a single class may hold multiple @Bean methods for the same bean, as an arrangement of multiple factory methods to use depending on available dependencies at runtime. This is the same algorithm as for choosing the "greediest" constructor or factory method in other configuration scenarios: The variant with the largest number of satisfiable dependencies will be picked at construction time, analogous to how the container selects between multiple @Autowired constructors.

1.10.6. Naming autodetected components

When a component is autodetected as part of the scanning process, its bean name is generated by the BeanNameGenerator strategy known to that scanner. By default, any Spring stereotype annotation (@Component, @Repository, @Service, and @Controller) that contains a name value will thereby provide that name to the corresponding bean definition.

If such an annotation contains no name value or for any other detected component (such as those discovered by custom filters), the default bean name generator returns the uncapitalized non-qualified class name. For example, if the following component classes were detected, the names would be myMovieLister and movieFinderImpl:

@Service("myMovieLister")

public class SimpleMovieLister {

// ...

}

@Repository

public class MovieFinderImpl implements MovieFinder {

// ...

}

If you do not want to rely on the default bean-naming strategy, you can provide a custom bean-naming strategy. First, implement the BeanNameGenerator interface, and be sure to include a default no-arg constructor. Then, provide the fully-qualified class name when configuring the scanner:

@Configuration

@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.class)

public class AppConfig {

...

}

<beans>

<context:component-scan base-package="org.example"

name-generator="org.example.MyNameGenerator" />

</beans>

As a general rule, consider specifying the name with the annotation whenever other components may be making explicit references to it. On the other hand, the auto-generated names are adequate whenever the container is responsible for wiring.

1.10.7. Providing a scope for autodetected components

As with Spring-managed components in general, the default and most common scope for autodetected components is singleton. However, sometimes you need a different scope which can be specified via the @Scope annotation. Simply provide the name of the scope within the annotation:

@Scope("prototype")

@Repository

public class MovieFinderImpl implements MovieFinder {

// ...

}

@Scope annotations are only introspected on the concrete bean class (for annotated components) or the factory method (for @Bean methods). In contrast to XML bean definitions, there is no notion of bean definition inheritance, and inheritance hierarchies at the class level are irrelevant for metadata purposes.

For details on web-specific scopes such as "request"/"session" in a Spring context, see Request, session, application, and WebSocket scopes. Like the pre-built annotations for those scopes, you may also compose your own scoping annotations using Spring’s meta-annotation approach: e.g. a custom annotation meta-annotated with @Scope("prototype"), possibly also declaring a custom scoped-proxy mode.

To provide a custom strategy for scope resolution rather than relying on the annotation-based approach, implement the ScopeMetadataResolver interface, and be sure to include a default no-arg constructor. Then, provide the fully-qualified class name when configuring the scanner:

@Configuration

@ComponentScan(basePackages = "org.example", scopeResolver = MyScopeResolver.class)

public class AppConfig {

...

}

<beans>

<context:component-scan base-package="org.example" scope-resolver="org.example.MyScopeResolver"/>

</beans>

When using certain non-singleton scopes, it may be necessary to generate proxies for the scoped objects. The reasoning is described in Scoped beans as dependencies. For this purpose, a scoped-proxy attribute is available on the component-scan element. The three possible values are: no, interfaces, and targetClass. For example, the following configuration will result in standard JDK dynamic proxies:

@Configuration

@ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode.INTERFACES)

public class AppConfig {

...

}

<beans>

<context:component-scan base-package="org.example" scoped-proxy="interfaces"/>

</beans>

1.10.8. Providing qualifier metadata with annotations

The @Qualifier annotation is discussed in Fine-tuning annotation-based autowiring with qualifiers. The examples in that section demonstrate the use of the @Qualifier annotation and custom qualifier annotations to provide fine-grained control when you resolve autowire candidates. Because those examples were based on XML bean definitions, the qualifier metadata was provided on the candidate bean definitions using the qualifier or meta sub-elements of the bean element in the XML. When relying upon classpath scanning for autodetection of components, you provide the qualifier metadata with type-level annotations on the candidate class. The following three examples demonstrate this technique:

@Component

@Qualifier("Action")

public class ActionMovieCatalog implements MovieCatalog {

// ...

}

@Component

@Genre("Action")

public class ActionMovieCatalog implements MovieCatalog {

// ...

}

@Component

@Offline

public class CachingMovieCatalog implements MovieCatalog {

// ...

}

As with most annotation-based alternatives, keep in mind that the annotation metadata is bound to the class definition itself, while the use of XML allows for multiple beans of the same type to provide variations in their qualifier metadata, because that metadata is provided per-instance rather than per-class.

1.10.9. Generating an index of candidate components

While classpath scanning is very fast, it is possible to improve the startup performance of large applications by creating a static list of candidates at compilation time. In this mode, all modules of the application must use this mechanism as, when the ApplicationContext detects such index, it will automatically use it rather than scanning the classpath.

To generate the index, simply add an additional dependency to each module that contains components that are target for component scan directives:

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-indexer</artifactId>

<version>5.0.8.RELEASE</version>

<optional>true</optional>

</dependency>

</dependencies>

Or, using Gradle:

dependencies {

compileOnly("org.springframework:spring-context-indexer:5.0.8.RELEASE")

}

That process will generate a META-INF/spring.components file that is going to be included in the jar.

When working with this mode in your IDE, the spring-context-indexer must be registered as an annotation processor to make sure the index is up to date when candidate components are updated.

The index is enabled automatically when a META-INF/spring.components is found on the classpath. If an index is partially available for some libraries (or use cases) but couldn’t be built for the whole application, you can fallback to a regular classpath arrangement (i.e. as no index was present at all) by setting spring.index.ignore to true, either as a system property or in a spring.properties file at the root of the classpath.

1.11. Using JSR 330 Standard Annotations

Starting with Spring 3.0, Spring offers support for JSR-330 standard annotations (Dependency Injection). Those annotations are scanned in the same way as the Spring annotations. You just need to have the relevant jars in your classpath.

If you are using Maven, the javax.inject artifact is available in the standard Maven repository ( http://repo1.maven.org/maven2/javax/inject/javax.inject/1/). You can add the following dependency to your file pom.xml:

<dependency>

<groupId>javax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>1</version>

</dependency>

1.11.1. Dependency Injection with @Inject and @Named

Instead of @Autowired, @javax.inject.Inject may be used as follows:

import javax.inject.Inject;

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Inject

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

public void listMovies() {

this.movieFinder.findMovies(...);

...

}

}

As with @Autowired, it is possible to use @Inject at the field level, method level and constructor-argument level. Furthermore, you may declare your injection point as a Provider, allowing for on-demand access to beans of shorter scopes or lazy access to other beans through a Provider.get() call. As a variant of the example above:

import javax.inject.Inject;

import javax.inject.Provider;

public class SimpleMovieLister {

private Provider<MovieFinder> movieFinder;

@Inject

public void setMovieFinder(Provider<MovieFinder> movieFinder) {

this.movieFinder = movieFinder;

}

public void listMovies() {

this.movieFinder.get().findMovies(...);

...

}

}

If you would like to use a qualified name for the dependency that should be injected, you should use the @Named annotation as follows:

import javax.inject.Inject;

import javax.inject.Named;

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Inject

public void setMovieFinder(@Named("main") MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

// ...

}

Like @Autowired, @Inject can also be used with java.util.Optional or @Nullable. This is even more applicable here since @Inject does not have a required attribute.

public class SimpleMovieLister {

@Inject

public void setMovieFinder(Optional<MovieFinder> movieFinder) {

...

}

}

public class SimpleMovieLister {

@Inject

public void setMovieFinder(@Nullable MovieFinder movieFinder) {

...

}

}

1.11.2. @Named and @ManagedBean: standard equivalents to the @Component annotation

Instead of @Component, @javax.inject.Named or javax.annotation.ManagedBean may be used as follows:

import javax.inject.Inject;

import javax.inject.Named;

@Named("movieListener") // @ManagedBean("movieListener") could be used as well

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Inject

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

// ...

}

It is very common to use @Component without specifying a name for the component. @Named can be used in a similar fashion:

import javax.inject.Inject;

import javax.inject.Named;

@Named

public class SimpleMovieLister {

private MovieFinder movieFinder;

@Inject

public void setMovieFinder(MovieFinder movieFinder) {

this.movieFinder = movieFinder;

}

// ...

}

When using @Named or @ManagedBean, it is possible to use component scanning in the exact same way as when using Spring annotations:

@Configuration

@ComponentScan(basePackages = "org.example")

public class AppConfig {

...

}

In contrast to @Component, the JSR-330 @Named and the JSR-250 ManagedBean annotations are not composable. Please use Spring’s stereotype model for building custom component annotations.

1.11.3. Limitations of JSR-330 standard annotations

When working with standard annotations, it is important to know that some significant features are not available as shown in the table below:

Table 6. Spring component model elements vs. JSR-330 variants

Spring javax.inject.\* javax.inject restrictions / comments

@Autowired

@Inject

@Inject has no 'required' attribute; can be used with Java 8’s Optional instead.

@Component

@Named / @ManagedBean

JSR-330 does not provide a composable model, just a way to identify named components.

@Scope("singleton")

@Singleton

The JSR-330 default scope is like Spring’s prototype. However, in order to keep it consistent with Spring’s general defaults, a JSR-330 bean declared in the Spring container is a singleton by default. In order to use a scope other than singleton, you should use Spring’s @Scope annotation. javax.inject also provides a @Scope annotation. Nevertheless, this one is only intended to be used for creating your own annotations.

@Qualifier

@Qualifier / @Named

javax.inject.Qualifier is just a meta-annotation for building custom qualifiers. Concrete String qualifiers (like Spring’s @Qualifier with a value) can be associated through javax.inject.Named.

@Value

-

no equivalent

@Required

-

no equivalent

@Lazy

-

no equivalent

ObjectFactory

Provider

javax.inject.Provider is a direct alternative to Spring’s ObjectFactory, just with a shorter get() method name. It can also be used in combination with Spring’s @Autowired or with non-annotated constructors and setter methods.

1.12. Java-based container configuration

1.12.1. Basic concepts: @Bean and @Configuration

The central artifacts in Spring’s new Java-configuration support are @Configuration-annotated classes and @Bean-annotated methods.

The @Bean annotation is used to indicate that a method instantiates, configures and initializes a new object to be managed by the Spring IoC container. For those familiar with Spring’s <beans/> XML configuration the @Bean annotation plays the same role as the <bean/> element. You can use @Bean annotated methods with any Spring @Component, however, they are most often used with @Configuration beans.

Annotating a class with @Configuration indicates that its primary purpose is as a source of bean definitions. Furthermore, @Configuration classes allow inter-bean dependencies to be defined by simply calling other @Bean methods in the same class. The simplest possible @Configuration class would read as follows:

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyServiceImpl();

}

}

The AppConfig class above would be equivalent to the following Spring <beans/> XML:

<beans>

<bean id="myService" class="com.acme.services.MyServiceImpl"/>

</beans>

Full @Configuration vs 'lite' @Bean mode?

When @Bean methods are declared within classes that are not annotated with @Configuration they are referred to as being processed in a 'lite' mode. Bean methods declared in a @Component or even in a plain old class will be considered 'lite', with a different primary purpose of the containing class and an @Bean method just being a sort of bonus there. For example, service components may expose management views to the container through an additional @Bean method on each applicable component class. In such scenarios, @Bean methods are a simple general-purpose factory method mechanism.

Unlike full @Configuration, lite @Bean methods cannot declare inter-bean dependencies. Instead, they operate on their containing component’s internal state and optionally on arguments that they may declare. Such an @Bean method should therefore not invoke other @Bean methods; each such method is literally just a factory method for a particular bean reference, without any special runtime semantics. The positive side-effect here is that no CGLIB subclassing has to be applied at runtime, so there are no limitations in terms of class design (i.e. the containing class may nevertheless be final etc).

In common scenarios, @Bean methods are to be declared within @Configuration classes, ensuring that 'full' mode is always used and that cross-method references will therefore get redirected to the container’s lifecycle management. This will prevent the same @Bean method from accidentally being invoked through a regular Java call which helps to reduce subtle bugs that can be hard to track down when operating in 'lite' mode.

The @Bean and @Configuration annotations will be discussed in depth in the sections below. First, however, we’ll cover the various ways of creating a spring container using Java-based configuration.

1.12.2. Instantiating the Spring container using AnnotationConfigApplicationContext

The sections below document Spring’s AnnotationConfigApplicationContext, new in Spring 3.0. This versatile ApplicationContext implementation is capable of accepting not only @Configuration classes as input, but also plain @Component classes and classes annotated with JSR-330 metadata.

When @Configuration classes are provided as input, the @Configuration class itself is registered as a bean definition, and all declared @Bean methods within the class are also registered as bean definitions.

When @Component and JSR-330 classes are provided, they are registered as bean definitions, and it is assumed that DI metadata such as @Autowired or @Inject are used within those classes where necessary.

Simple construction

In much the same way that Spring XML files are used as input when instantiating a ClassPathXmlApplicationContext, @Configuration classes may be used as input when instantiating an AnnotationConfigApplicationContext. This allows for completely XML-free usage of the Spring container:

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

As mentioned above, AnnotationConfigApplicationContext is not limited to working only with @Configuration classes. Any @Component or JSR-330 annotated class may be supplied as input to the constructor. For example:

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(MyServiceImpl.class, Dependency1.class, Dependency2.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

The above assumes that MyServiceImpl, Dependency1 and Dependency2 use Spring dependency injection annotations such as @Autowired.

Building the container programmatically using register(Class<?>…​)

An AnnotationConfigApplicationContext may be instantiated using a no-arg constructor and then configured using the register() method. This approach is particularly useful when programmatically building an AnnotationConfigApplicationContext.

public static void main(String[] args) {

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.register(AppConfig.class, OtherConfig.class);

ctx.register(AdditionalConfig.class);

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();

}

Enabling component scanning with scan(String…​)

To enable component scanning, just annotate your @Configuration class as follows:

@Configuration

@ComponentScan(basePackages = "com.acme")

public class AppConfig {

...

}

Experienced Spring users will be familiar with the XML declaration equivalent from Spring’s context: namespace

<beans>

<context:component-scan base-package="com.acme"/>

</beans>

In the example above, the com.acme package will be scanned, looking for any @Component-annotated classes, and those classes will be registered as Spring bean definitions within the container. AnnotationConfigApplicationContext exposes the scan(String…​) method to allow for the same component-scanning functionality:

public static void main(String[] args) {

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.scan("com.acme");

ctx.refresh();

MyService myService = ctx.getBean(MyService.class);

}

Remember that @Configuration classes are meta-annotated with @Component, so they are candidates for component-scanning! In the example above, assuming that AppConfig is declared within the com.acme package (or any package underneath), it will be picked up during the call to scan(), and upon refresh() all its @Bean methods will be processed and registered as bean definitions within the container.

Support for web applications with AnnotationConfigWebApplicationContext

A WebApplicationContext variant of AnnotationConfigApplicationContext is available with AnnotationConfigWebApplicationContext. This implementation may be used when configuring the Spring ContextLoaderListener servlet listener, Spring MVC DispatcherServlet, etc. What follows is a web.xml snippet that configures a typical Spring MVC web application. Note the use of the contextClass context-param and init-param:

<web-app>

<!-- Configure ContextLoaderListener to use AnnotationConfigWebApplicationContext

instead of the default XmlWebApplicationContext -->

<context-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</context-param>

<!-- Configuration locations must consist of one or more comma- or space-delimited

fully-qualified @Configuration classes. Fully-qualified packages may also be

specified for component-scanning -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.acme.AppConfig</param-value>

</context-param>

<!-- Bootstrap the root application context as usual using ContextLoaderListener -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!-- Declare a Spring MVC DispatcherServlet as usual -->

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<!-- Configure DispatcherServlet to use AnnotationConfigWebApplicationContext

instead of the default XmlWebApplicationContext -->

<init-param>

<param-name>contextClass</param-name>

<param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext

</param-value>

</init-param>

<!-- Again, config locations must consist of one or more comma- or space-delimited

and fully-qualified @Configuration classes -->

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.acme.web.MvcConfig</param-value>

</init-param>

</servlet>

<!-- map all requests for /app/\* to the dispatcher servlet -->

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/app/\*</url-pattern>

</servlet-mapping>

</web-app>

1.12.3. Using the @Bean annotation

@Bean is a method-level annotation and a direct analog of the XML <bean/> element. The annotation supports some of the attributes offered by <bean/>, such as: init-method, destroy-method, autowiring and name.

You can use the @Bean annotation in a @Configuration-annotated or in a @Component-annotated class.

Declaring a bean

To declare a bean, simply annotate a method with the @Bean annotation. You use this method to register a bean definition within an ApplicationContext of the type specified as the method’s return value. By default, the bean name will be the same as the method name. The following is a simple example of a @Bean method declaration:

@Configuration

public class AppConfig {

@Bean

public TransferServiceImpl transferService() {

return new TransferServiceImpl();

}

}

The preceding configuration is exactly equivalent to the following Spring XML:

<beans>

<bean id="transferService" class="com.acme.TransferServiceImpl"/>

</beans>

Both declarations make a bean named transferService available in the ApplicationContext, bound to an object instance of type TransferServiceImpl:

transferService -> com.acme.TransferServiceImpl

You may also declare your @Bean method with an interface (or base class) return type:

@Configuration

public class AppConfig {

@Bean

public TransferService transferService() {

return new TransferServiceImpl();

}

}

However, this limits the visibility for advance type prediction to the specified interface type (TransferService) then, with the full type (TransferServiceImpl) only known to the container once the affected singleton bean has been instantiated. Non-lazy singleton beans get instantiated according to their declaration order, so you may see different type matching results depending on when another component tries to match by a non-declared type (such as @Autowired TransferServiceImpl which will only resolve once the "transferService" bean has been instantiated).

If you consistently refer to your types by a declared service interface, your @Bean return types may safely join that design decision. However, for components implementing several interfaces or for components potentially referred to by their implementation type, it is safer to declare the most specific return type possible (at least as specific as required by the injection points referring to your bean).

Bean dependencies

A @Bean annotated method can have an arbitrary number of parameters describing the dependencies required to build that bean. For instance if our TransferService requires an AccountRepository we can materialize that dependency via a method parameter:

@Configuration

public class AppConfig {

@Bean

public TransferService transferService(AccountRepository accountRepository) {

return new TransferServiceImpl(accountRepository);

}

}

The resolution mechanism is pretty much identical to constructor-based dependency injection, see the relevant section for more details.

Receiving lifecycle callbacks

Any classes defined with the @Bean annotation support the regular lifecycle callbacks and can use the @PostConstruct and @PreDestroy annotations from JSR-250, see JSR-250 annotations for further details.

The regular Spring lifecycle callbacks are fully supported as well. If a bean implements InitializingBean, DisposableBean, or Lifecycle, their respective methods are called by the container.

The standard set of \*Aware interfaces such as BeanFactoryAware, BeanNameAware, MessageSourceAware, ApplicationContextAware, and so on are also fully supported.

The @Bean annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML’s init-method and destroy-method attributes on the bean element:

public class Foo {

public void init() {

// initialization logic

}

}

public class Bar {

public void cleanup() {

// destruction logic

}

}

@Configuration

public class AppConfig {

@Bean(initMethod = "init")

public Foo foo() {

return new Foo();

}

@Bean(destroyMethod = "cleanup")

public Bar bar() {

return new Bar();

}

}

By default, beans defined using Java config that have a public close or shutdown method are automatically enlisted with a destruction callback. If you have a public close or shutdown method and you do not wish for it to be called when the container shuts down, simply add @Bean(destroyMethod="") to your bean definition to disable the default (inferred) mode.

You may want to do that by default for a resource that you acquire via JNDI as its lifecycle is managed outside the application. In particular, make sure to always do it for a DataSource as it is known to be problematic on Java EE application servers.

@Bean(destroyMethod="")

public DataSource dataSource() throws NamingException {

return (DataSource) jndiTemplate.lookup("MyDS");

}

Also, with @Bean methods, you will typically choose to use programmatic JNDI lookups: either using Spring’s JndiTemplate/JndiLocatorDelegate helpers or straight JNDI InitialContext usage, but not the JndiObjectFactoryBean variant which would force you to declare the return type as the FactoryBean type instead of the actual target type, making it harder to use for cross-reference calls in other @Bean methods that intend to refer to the provided resource here.

Of course, in the case of Foo above, it would be equally as valid to call the init() method directly during construction:

@Configuration

public class AppConfig {

@Bean

public Foo foo() {

Foo foo = new Foo();

foo.init();

return foo;

}

// ...

}

When you work directly in Java, you can do anything you like with your objects and do not always need to rely on the container lifecycle!

Specifying bean scope

Using the @Scope annotation

You can specify that your beans defined with the @Bean annotation should have a specific scope. You can use any of the standard scopes specified in the Bean Scopes section.

The default scope is singleton, but you can override this with the @Scope annotation:

@Configuration

public class MyConfiguration {

@Bean

@Scope("prototype")

public Encryptor encryptor() {

// ...

}

}

@Scope and scoped-proxy

Spring offers a convenient way of working with scoped dependencies through scoped proxies. The easiest way to create such a proxy when using the XML configuration is the <aop:scoped-proxy/> element. Configuring your beans in Java with a @Scope annotation offers equivalent support with the proxyMode attribute. The default is no proxy ( ScopedProxyMode.NO), but you can specify ScopedProxyMode.TARGET\_CLASS or ScopedProxyMode.INTERFACES.

If you port the scoped proxy example from the XML reference documentation (see preceding link) to our @Bean using Java, it would look like the following:

// an HTTP Session-scoped bean exposed as a proxy

@Bean

@SessionScope

public UserPreferences userPreferences() {

return new UserPreferences();

}

@Bean

public Service userService() {

UserService service = new SimpleUserService();

// a reference to the proxied userPreferences bean

service.setUserPreferences(userPreferences());

return service;

}

Customizing bean naming

By default, configuration classes use a @Bean method’s name as the name of the resulting bean. This functionality can be overridden, however, with the name attribute.

@Configuration

public class AppConfig {

@Bean(name = "myFoo")

public Foo foo() {

return new Foo();

}

}

Bean aliasing

As discussed in Naming beans, it is sometimes desirable to give a single bean multiple names, otherwise known as bean aliasing. The name attribute of the @Bean annotation accepts a String array for this purpose.

@Configuration

public class AppConfig {

@Bean(name = { "dataSource", "subsystemA-dataSource", "subsystemB-dataSource" })

public DataSource dataSource() {

// instantiate, configure and return DataSource bean...

}

}

Bean description

Sometimes it is helpful to provide a more detailed textual description of a bean. This can be particularly useful when beans are exposed (perhaps via JMX) for monitoring purposes.

To add a description to a @Bean the @Description annotation can be used:

@Configuration

public class AppConfig {

@Bean

@Description("Provides a basic example of a bean")

public Foo foo() {

return new Foo();

}

}

1.12.4. Using the @Configuration annotation

@Configuration is a class-level annotation indicating that an object is a source of bean definitions. @Configuration classes declare beans via public @Bean annotated methods. Calls to @Bean methods on @Configuration classes can also be used to define inter-bean dependencies. See Basic concepts: @Bean and @Configuration for a general introduction.

Injecting inter-bean dependencies

When @Beans have dependencies on one another, expressing that dependency is as simple as having one bean method call another:

@Configuration

public class AppConfig {

@Bean

public Foo foo() {

return new Foo(bar());

}

@Bean

public Bar bar() {

return new Bar();

}

}

In the example above, the foo bean receives a reference to bar via constructor injection.

This method of declaring inter-bean dependencies only works when the @Bean method is declared within a @Configuration class. You cannot declare inter-bean dependencies using plain @Component classes.

Lookup method injection

As noted earlier, lookup method injection is an advanced feature that you should use rarely. It is useful in cases where a singleton-scoped bean has a dependency on a prototype-scoped bean. Using Java for this type of configuration provides a natural means for implementing this pattern.

public abstract class CommandManager {

public Object process(Object commandState) {

// grab a new instance of the appropriate Command interface

Command command = createCommand();

// set the state on the (hopefully brand new) Command instance

command.setState(commandState);

return command.execute();

}

// okay... but where is the implementation of this method?

protected abstract Command createCommand();

}

Using Java-configuration support , you can create a subclass of CommandManager where the abstract createCommand() method is overridden in such a way that it looks up a new (prototype) command object:

@Bean

@Scope("prototype")

public AsyncCommand asyncCommand() {

AsyncCommand command = new AsyncCommand();

// inject dependencies here as required

return command;

}

@Bean

public CommandManager commandManager() {

// return new anonymous implementation of CommandManager with command() overridden

// to return a new prototype Command object

return new CommandManager() {

protected Command createCommand() {

return asyncCommand();

}

}

}

Further information about how Java-based configuration works internally

The following example shows a @Bean annotated method being called twice:

@Configuration

public class AppConfig {

@Bean

public ClientService clientService1() {

ClientServiceImpl clientService = new ClientServiceImpl();

clientService.setClientDao(clientDao());

return clientService;

}

@Bean

public ClientService clientService2() {

ClientServiceImpl clientService = new ClientServiceImpl();

clientService.setClientDao(clientDao());

return clientService;

}

@Bean

public ClientDao clientDao() {

return new ClientDaoImpl();

}

}

clientDao() has been called once in clientService1() and once in clientService2(). Since this method creates a new instance of ClientDaoImpl and returns it, you would normally expect having 2 instances (one for each service). That definitely would be problematic: in Spring, instantiated beans have a singleton scope by default. This is where the magic comes in: All @Configuration classes are subclassed at startup-time with CGLIB. In the subclass, the child method checks the container first for any cached (scoped) beans before it calls the parent method and creates a new instance. Note that as of Spring 3.2, it is no longer necessary to add CGLIB to your classpath because CGLIB classes have been repackaged under org.springframework.cglib and included directly within the spring-core JAR.

The behavior could be different according to the scope of your bean. We are talking about singletons here.

There are a few restrictions due to the fact that CGLIB dynamically adds features at startup-time, in particular that configuration classes must not be final. However, as of 4.3, any constructors are allowed on configuration classes, including the use of @Autowired or a single non-default constructor declaration for default injection.

If you prefer to avoid any CGLIB-imposed limitations, consider declaring your @Bean methods on non-@Configuration classes, e.g. on plain @Component classes instead. Cross-method calls between @Bean methods won’t get intercepted then, so you’ll have to exclusively rely on dependency injection at the constructor or method level there.

1.12.5. Composing Java-based configurations

Using the @Import annotation

Much as the <import/> element is used within Spring XML files to aid in modularizing configurations, the @Import annotation allows for loading @Bean definitions from another configuration class:

@Configuration

public class ConfigA {

@Bean

public A a() {

return new A();

}

}

@Configuration

@Import(ConfigA.class)

public class ConfigB {

@Bean

public B b() {

return new B();

}

}

Now, rather than needing to specify both ConfigA.class and ConfigB.class when instantiating the context, only ConfigB needs to be supplied explicitly:

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class);

// now both beans A and B will be available...

A a = ctx.getBean(A.class);

B b = ctx.getBean(B.class);

}

This approach simplifies container instantiation, as only one class needs to be dealt with, rather than requiring the developer to remember a potentially large number of @Configuration classes during construction.

As of Spring Framework 4.2, @Import also supports references to regular component classes, analogous to the AnnotationConfigApplicationContext.register method. This is particularly useful if you’d like to avoid component scanning, using a few configuration classes as entry points for explicitly defining all your components.

Injecting dependencies on imported @Bean definitions

The example above works, but is simplistic. In most practical scenarios, beans will have dependencies on one another across configuration classes. When using XML, this is not an issue, per se, because there is no compiler involved, and one can simply declare ref="someBean" and trust that Spring will work it out during container initialization. Of course, when using @Configuration classes, the Java compiler places constraints on the configuration model, in that references to other beans must be valid Java syntax.

Fortunately, solving this problem is simple. As we already discussed, @Bean method can have an arbitrary number of parameters describing the bean dependencies. Let’s consider a more real-world scenario with several @Configuration classes, each depending on beans declared in the others:

@Configuration

public class ServiceConfig {

@Bean

public TransferService transferService(AccountRepository accountRepository) {

return new TransferServiceImpl(accountRepository);

}

}

@Configuration

public class RepositoryConfig {

@Bean

public AccountRepository accountRepository(DataSource dataSource) {

return new JdbcAccountRepository(dataSource);

}

}

@Configuration

@Import({ServiceConfig.class, RepositoryConfig.class})

public class SystemTestConfig {

@Bean

public DataSource dataSource() {

// return new DataSource

}

}

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(SystemTestConfig.class);

// everything wires up across configuration classes...

TransferService transferService = ctx.getBean(TransferService.class);

transferService.transfer(100.00, "A123", "C456");

}

There is another way to achieve the same result. Remember that @Configuration classes are ultimately just another bean in the container: This means that they can take advantage of @Autowired and @Value injection etc just like any other bean!

Make sure that the dependencies you inject that way are of the simplest kind only. @Configuration classes are processed quite early during the initialization of the context and forcing a dependency to be injected this way may lead to unexpected early initialization. Whenever possible, resort to parameter-based injection as in the example above.

Also, be particularly careful with BeanPostProcessor and BeanFactoryPostProcessor definitions via @Bean. Those should usually be declared as static @Bean methods, not triggering the instantiation of their containing configuration class. Otherwise, @Autowired and @Value won’t work on the configuration class itself since it is being created as a bean instance too early.

@Configuration

public class ServiceConfig {

@Autowired

private AccountRepository accountRepository;

@Bean

public TransferService transferService() {

return new TransferServiceImpl(accountRepository);

}

}

@Configuration

public class RepositoryConfig {

private final DataSource dataSource;

@Autowired

public RepositoryConfig(DataSource dataSource) {

this.dataSource = dataSource;

}

@Bean

public AccountRepository accountRepository() {

return new JdbcAccountRepository(dataSource);

}

}

@Configuration

@Import({ServiceConfig.class, RepositoryConfig.class})

public class SystemTestConfig {

@Bean

public DataSource dataSource() {

// return new DataSource

}

}

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(SystemTestConfig.class);

// everything wires up across configuration classes...

TransferService transferService = ctx.getBean(TransferService.class);

transferService.transfer(100.00, "A123", "C456");

}

Constructor injection in @Configuration classes is only supported as of Spring Framework 4.3. Note also that there is no need to specify @Autowired if the target bean defines only one constructor; in the example above, @Autowired is not necessary on the RepositoryConfig constructor.

Fully-qualifying imported beans for ease of navigation

In the scenario above, using @Autowired works well and provides the desired modularity, but determining exactly where the autowired bean definitions are declared is still somewhat ambiguous. For example, as a developer looking at ServiceConfig, how do you know exactly where the @Autowired AccountRepository bean is declared? It’s not explicit in the code, and this may be just fine. Remember that the Spring Tool Suite provides tooling that can render graphs showing how everything is wired up - that may be all you need. Also, your Java IDE can easily find all declarations and uses of the AccountRepository type, and will quickly show you the location of @Bean methods that return that type.

In cases where this ambiguity is not acceptable and you wish to have direct navigation from within your IDE from one @Configuration class to another, consider autowiring the configuration classes themselves:

@Configuration

public class ServiceConfig {

@Autowired

private RepositoryConfig repositoryConfig;

@Bean

public TransferService transferService() {

// navigate 'through' the config class to the @Bean method!

return new TransferServiceImpl(repositoryConfig.accountRepository());

}

}

In the situation above, it is completely explicit where AccountRepository is defined. However, ServiceConfig is now tightly coupled to RepositoryConfig; that’s the tradeoff. This tight coupling can be somewhat mitigated by using interface-based or abstract class-based @Configuration classes. Consider the following:

@Configuration

public class ServiceConfig {

@Autowired

private RepositoryConfig repositoryConfig;

@Bean

public TransferService transferService() {

return new TransferServiceImpl(repositoryConfig.accountRepository());

}

}

@Configuration

public interface RepositoryConfig {

@Bean

AccountRepository accountRepository();

}

@Configuration

public class DefaultRepositoryConfig implements RepositoryConfig {

@Bean

public AccountRepository accountRepository() {

return new JdbcAccountRepository(...);

}

}

@Configuration

@Import({ServiceConfig.class, DefaultRepositoryConfig.class}) // import the concrete config!

public class SystemTestConfig {

@Bean

public DataSource dataSource() {

// return DataSource

}

}

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(SystemTestConfig.class);

TransferService transferService = ctx.getBean(TransferService.class);

transferService.transfer(100.00, "A123", "C456");

}

Now ServiceConfig is loosely coupled with respect to the concrete DefaultRepositoryConfig, and built-in IDE tooling is still useful: it will be easy for the developer to get a type hierarchy of RepositoryConfig implementations. In this way, navigating @Configuration classes and their dependencies becomes no different than the usual process of navigating interface-based code.

If you would like to influence the startup creation order of certain beans, consider declaring some of them as @Lazy (for creation on first access instead of on startup) or as @DependsOn on certain other beans (making sure that specific other beans will be created before the current bean, beyond what the latter’s direct dependencies imply).

Conditionally include @Configuration classes or @Bean methods

It is often useful to conditionally enable or disable a complete @Configuration class, or even individual @Bean methods, based on some arbitrary system state. One common example of this is to use the @Profile annotation to activate beans only when a specific profile has been enabled in the Spring Environment (see Bean definition profiles for details).

The @Profile annotation is actually implemented using a much more flexible annotation called @Conditional. The @Conditional annotation indicates specific org.springframework.context.annotation.Condition implementations that should be consulted before a @Bean is registered.

Implementations of the Condition interface simply provide a matches(…​) method that returns true or false. For example, here is the actual Condition implementation used for @Profile:

@Override

public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {

if (context.getEnvironment() != null) {

// Read the @Profile annotation attributes

MultiValueMap<String, Object> attrs = metadata.getAllAnnotationAttributes(Profile.class.getName());

if (attrs != null) {

for (Object value : attrs.get("value")) {

if (context.getEnvironment().acceptsProfiles(((String[]) value))) {

return true;

}

}

return false;

}

}

return true;

}

See the @Conditional javadocs for more detail.

Combining Java and XML configuration

Spring’s @Configuration class support does not aim to be a 100% complete replacement for Spring XML. Some facilities such as Spring XML namespaces remain an ideal way to configure the container. In cases where XML is convenient or necessary, you have a choice: either instantiate the container in an "XML-centric" way using, for example, ClassPathXmlApplicationContext, or in a "Java-centric" fashion using AnnotationConfigApplicationContext and the @ImportResource annotation to import XML as needed.

XML-centric use of @Configuration classes

It may be preferable to bootstrap the Spring container from XML and include @Configuration classes in an ad-hoc fashion. For example, in a large existing codebase that uses Spring XML, it will be easier to create @Configuration classes on an as-needed basis and include them from the existing XML files. Below you’ll find the options for using @Configuration classes in this kind of "XML-centric" situation.

Declaring @Configuration classes as plain Spring <bean/> elements

Remember that @Configuration classes are ultimately just bean definitions in the container. In this example, we create a @Configuration class named AppConfig and include it within system-test-config.xml as a <bean/> definition. Because <context:annotation-config/> is switched on, the container will recognize the @Configuration annotation and process the @Bean methods declared in AppConfig properly.

@Configuration

public class AppConfig {

@Autowired

private DataSource dataSource;

@Bean

public AccountRepository accountRepository() {

return new JdbcAccountRepository(dataSource);

}

@Bean

public TransferService transferService() {

return new TransferService(accountRepository());

}

}

system-test-config.xml:

<beans>

<!-- enable processing of annotations such as @Autowired and @Configuration -->

<context:annotation-config/>

<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

<bean class="com.acme.AppConfig"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

</beans>

jdbc.properties:

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

public static void main(String[] args) {

ApplicationContext ctx = new ClassPathXmlApplicationContext("classpath:/com/acme/system-test-config.xml");

TransferService transferService = ctx.getBean(TransferService.class);

// ...

}

In system-test-config.xml above, the AppConfig <bean/> does not declare an id element. While it would be acceptable to do so, it is unnecessary given that no other bean will ever refer to it, and it is unlikely that it will be explicitly fetched from the container by name. Likewise with the DataSource bean - it is only ever autowired by type, so an explicit bean id is not strictly required.

Using <context:component-scan/> to pick up @Configuration classes

Because @Configuration is meta-annotated with @Component, @Configuration-annotated classes are automatically candidates for component scanning. Using the same scenario as above, we can redefine system-test-config.xml to take advantage of component-scanning. Note that in this case, we don’t need to explicitly declare <context:annotation-config/>, because <context:component-scan/> enables the same functionality.

system-test-config.xml:

<beans>

<!-- picks up and registers AppConfig as a bean definition -->

<context:component-scan base-package="com.acme"/>

<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

</beans>

@Configuration class-centric use of XML with @ImportResource

In applications where @Configuration classes are the primary mechanism for configuring the container, it will still likely be necessary to use at least some XML. In these scenarios, simply use @ImportResource and define only as much XML as is needed. Doing so achieves a "Java-centric" approach to configuring the container and keeps XML to a bare minimum.

@Configuration

@ImportResource("classpath:/com/acme/properties-config.xml")

public class AppConfig {

@Value("${jdbc.url}")

private String url;

@Value("${jdbc.username}")

private String username;

@Value("${jdbc.password}")

private String password;

@Bean

public DataSource dataSource() {

return new DriverManagerDataSource(url, username, password);

}

}

properties-config.xml

<beans>

<context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

</beans>

jdbc.properties

jdbc.url=jdbc:hsqldb:hsql://localhost/xdb

jdbc.username=sa

jdbc.password=

public static void main(String[] args) {

ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

TransferService transferService = ctx.getBean(TransferService.class);

// ...

}

1.13. Environment abstraction

The Environment is an abstraction integrated in the container that models two key aspects of the application environment: profiles and properties.

A profile is a named, logical group of bean definitions to be registered with the container only if the given profile is active. Beans may be assigned to a profile whether defined in XML or via annotations. The role of the Environment object with relation to profiles is in determining which profiles (if any) are currently active, and which profiles (if any) should be active by default.

Properties play an important role in almost all applications, and may originate from a variety of sources: properties files, JVM system properties, system environment variables, JNDI, servlet context parameters, ad-hoc Properties objects, Maps, and so on. The role of the Environment object with relation to properties is to provide the user with a convenient service interface for configuring property sources and resolving properties from them.

1.13.1. Bean definition profiles

Bean definition profiles is a mechanism in the core container that allows for registration of different beans in different environments. The word environment can mean different things to different users and this feature can help with many use cases, including:

working against an in-memory datasource in development vs looking up that same datasource from JNDI when in QA or production

registering monitoring infrastructure only when deploying an application into a performance environment

registering customized implementations of beans for customer A vs. customer B deployments

Let’s consider the first use case in a practical application that requires a DataSource. In a test environment, the configuration may look like this:

@Bean

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("my-schema.sql")

.addScript("my-test-data.sql")

.build();

}

Let’s now consider how this application will be deployed into a QA or production environment, assuming that the datasource for the application will be registered with the production application server’s JNDI directory. Our dataSource bean now looks like this:

@Bean(destroyMethod="")

public DataSource dataSource() throws Exception {

Context ctx = new InitialContext();

return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");

}

The problem is how to switch between using these two variations based on the current environment. Over time, Spring users have devised a number of ways to get this done, usually relying on a combination of system environment variables and XML <import/> statements containing ${placeholder} tokens that resolve to the correct configuration file path depending on the value of an environment variable. Bean definition profiles is a core container feature that provides a solution to this problem.

If we generalize the example use case above of environment-specific bean definitions, we end up with the need to register certain bean definitions in certain contexts, while not in others. You could say that you want to register a certain profile of bean definitions in situation A, and a different profile in situation B. Let’s first see how we can update our configuration to reflect this need.

@Profile

The @Profile annotation allows you to indicate that a component is eligible for registration when one or more specified profiles are active. Using our example above, we can rewrite the dataSource configuration as follows:

@Configuration

@Profile("development")

public class StandaloneDataConfig {

@Bean

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.addScript("classpath:com/bank/config/sql/test-data.sql")

.build();

}

}

@Configuration

@Profile("production")

public class JndiDataConfig {

@Bean(destroyMethod="")

public DataSource dataSource() throws Exception {

Context ctx = new InitialContext();

return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");

}

}

As mentioned before, with @Bean methods, you will typically choose to use programmatic JNDI lookups: either using Spring’s JndiTemplate/JndiLocatorDelegate helpers or the straight JNDI InitialContext usage shown above, but not the JndiObjectFactoryBean variant which would force you to declare the return type as the FactoryBean type.

@Profile can be used as a meta-annotation for the purpose of creating a custom composed annotation. The following example defines a custom @Production annotation that can be used as a drop-in replacement for @Profile("production"):

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Profile("production")

public @interface Production {

}

If a @Configuration class is marked with @Profile, all of the @Bean methods and @Import annotations associated with that class will be bypassed unless one or more of the specified profiles are active. If a @Component or @Configuration class is marked with @Profile({"p1", "p2"}), that class will not be registered/processed unless profiles 'p1' and/or 'p2' have been activated. If a given profile is prefixed with the NOT operator (!), the annotated element will be registered if the profile is not active. For example, given @Profile({"p1", "!p2"}), registration will occur if profile 'p1' is active or if profile 'p2' is not active.

@Profile can also be declared at the method level to include only one particular bean of a configuration class, e.g. for alternative variants of a particular bean:

@Configuration

public class AppConfig {

@Bean("dataSource")

@Profile("development")

public DataSource standaloneDataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.addScript("classpath:com/bank/config/sql/test-data.sql")

.build();

}

@Bean("dataSource")

@Profile("production")

public DataSource jndiDataSource() throws Exception {

Context ctx = new InitialContext();

return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");

}

}

With @Profile on @Bean methods, a special scenario may apply: In the case of overloaded @Bean methods of the same Java method name (analogous to constructor overloading), an @Profile condition needs to be consistently declared on all overloaded methods. If the conditions are inconsistent, only the condition on the first declaration among the overloaded methods will matter. @Profile can therefore not be used to select an overloaded method with a particular argument signature over another; resolution between all factory methods for the same bean follows Spring’s constructor resolution algorithm at creation time.

If you would like to define alternative beans with different profile conditions, use distinct Java method names pointing to the same bean name via the @Bean name attribute, as indicated in the example above. If the argument signatures are all the same (e.g. all of the variants have no-arg factory methods), this is the only way to represent such an arrangement in a valid Java class in the first place (since there can only be one method of a particular name and argument signature).

XML bean definition profiles

The XML counterpart is the profile attribute of the <beans> element. Our sample configuration above can be rewritten in two XML files as follows:

<beans profile="development"

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:jdbc="http://www.springframework.org/schema/jdbc"

xsi:schemaLocation="...">

<jdbc:embedded-database id="dataSource">

<jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>

<jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>

</jdbc:embedded-database>

</beans>

<beans profile="production"

xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:jee="http://www.springframework.org/schema/jee"

xsi:schemaLocation="...">

<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>

</beans>

It is also possible to avoid that split and nest <beans/> elements within the same file:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:jdbc="http://www.springframework.org/schema/jdbc"

xmlns:jee="http://www.springframework.org/schema/jee"

xsi:schemaLocation="...">

<!-- other bean definitions -->

<beans profile="development">

<jdbc:embedded-database id="dataSource">

<jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>

<jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>

</jdbc:embedded-database>

</beans>

<beans profile="production">

<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>

</beans>

</beans>

The spring-bean.xsd has been constrained to allow such elements only as the last ones in the file. This should help provide flexibility without incurring clutter in the XML files.

Activating a profile

Now that we have updated our configuration, we still need to instruct Spring which profile is active. If we started our sample application right now, we would see a NoSuchBeanDefinitionException thrown, because the container could not find the Spring bean named dataSource.

Activating a profile can be done in several ways, but the most straightforward is to do it programmatically against the Environment API which is available via an ApplicationContext:

AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();

ctx.getEnvironment().setActiveProfiles("development");

ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);

ctx.refresh();

In addition, profiles may also be activated declaratively through the spring.profiles.active property which may be specified through system environment variables, JVM system properties, servlet context parameters in web.xml, or even as an entry in JNDI (see PropertySource abstraction). In integration tests, active profiles can be declared via the @ActiveProfiles annotation in the spring-test module (see Context configuration with environment profiles).

Note that profiles are not an "either-or" proposition; it is possible to activate multiple profiles at once. Programmatically, simply provide multiple profile names to the setActiveProfiles() method, which accepts String…​ varargs:

ctx.getEnvironment().setActiveProfiles("profile1", "profile2");

Declaratively, spring.profiles.active may accept a comma-separated list of profile names:

-Dspring.profiles.active="profile1,profile2"

Default profile

The default profile represents the profile that is enabled by default. Consider the following:

@Configuration

@Profile("default")

public class DefaultDataConfig {

@Bean

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.build();

}

}

If no profile is active, the dataSource above will be created; this can be seen as a way to provide a default definition for one or more beans. If any profile is enabled, the default profile will not apply.

The name of the default profile can be changed using setDefaultProfiles() on the Environment or declaratively using the spring.profiles.default property.

1.13.2. PropertySource abstraction

Spring’s Environment abstraction provides search operations over a configurable hierarchy of property sources. To explain fully, consider the following:

ApplicationContext ctx = new GenericApplicationContext();

Environment env = ctx.getEnvironment();

boolean containsFoo = env.containsProperty("foo");

System.out.println("Does my environment contain the 'foo' property? " + containsFoo);

In the snippet above, we see a high-level way of asking Spring whether the foo property is defined for the current environment. To answer this question, the Environment object performs a search over a set of PropertySource objects. A PropertySource is a simple abstraction over any source of key-value pairs, and Spring’s StandardEnvironment is configured with two PropertySource objects — one representing the set of JVM system properties (a la System.getProperties()) and one representing the set of system environment variables (a la System.getenv()).

These default property sources are present for StandardEnvironment, for use in standalone applications. StandardServletEnvironment is populated with additional default property sources including servlet config and servlet context parameters. It can optionally enable a JndiPropertySource. See the javadocs for details.

Concretely, when using the StandardEnvironment, the call to env.containsProperty("foo") will return true if a foo system property or foo environment variable is present at runtime.

The search performed is hierarchical. By default, system properties have precedence over environment variables, so if the foo property happens to be set in both places during a call to env.getProperty("foo"), the system property value will 'win' and be returned preferentially over the environment variable. Note that property values will not get merged but rather completely overridden by a preceding entry.

For a common StandardServletEnvironment, the full hierarchy looks as follows, with the highest-precedence entries at the top:

ServletConfig parameters (if applicable, e.g. in case of a DispatcherServlet context)

ServletContext parameters (web.xml context-param entries)

JNDI environment variables ("java:comp/env/" entries)

JVM system properties ("-D" command-line arguments)

JVM system environment (operating system environment variables)

Most importantly, the entire mechanism is configurable. Perhaps you have a custom source of properties that you’d like to integrate into this search. No problem — simply implement and instantiate your own PropertySource and add it to the set of PropertySources for the current Environment:

ConfigurableApplicationContext ctx = new GenericApplicationContext();

MutablePropertySources sources = ctx.getEnvironment().getPropertySources();

sources.addFirst(new MyPropertySource());

In the code above, MyPropertySource has been added with highest precedence in the search. If it contains a foo property, it will be detected and returned ahead of any foo property in any other PropertySource. The MutablePropertySources API exposes a number of methods that allow for precise manipulation of the set of property sources.

1.13.3. @PropertySource

The @PropertySource annotation provides a convenient and declarative mechanism for adding a PropertySource to Spring’s Environment.

Given a file "app.properties" containing the key/value pair testbean.name=myTestBean, the following @Configuration class uses @PropertySource in such a way that a call to testBean.getName() will return "myTestBean".

@Configuration

@PropertySource("classpath:/com/myco/app.properties")

public class AppConfig {

@Autowired

Environment env;

@Bean

public TestBean testBean() {

TestBean testBean = new TestBean();

testBean.setName(env.getProperty("testbean.name"));

return testBean;

}

}

Any ${…​} placeholders present in a @PropertySource resource location will be resolved against the set of property sources already registered against the environment. For example:

@Configuration

@PropertySource("classpath:/com/${my.placeholder:default/path}/app.properties")

public class AppConfig {

@Autowired

Environment env;

@Bean

public TestBean testBean() {

TestBean testBean = new TestBean();

testBean.setName(env.getProperty("testbean.name"));

return testBean;

}

}

Assuming that "my.placeholder" is present in one of the property sources already registered, e.g. system properties or environment variables, the placeholder will be resolved to the corresponding value. If not, then "default/path" will be used as a default. If no default is specified and a property cannot be resolved, an IllegalArgumentException will be thrown.

The @PropertySource annotation is repeatable according to Java 8 conventions. However, all such @PropertySource annotations need to be declared at the same level: either directly on the configuration class or as meta-annotations within the same custom annotation. Mixing of direct annotations and meta-annotations is not recommended since direct annotations will effectively override meta-annotations.

1.13.4. Placeholder resolution in statements

Historically, the value of placeholders in elements could be resolved only against JVM system properties or environment variables. No longer is this the case. Because the Environment abstraction is integrated throughout the container, it’s easy to route resolution of placeholders through it. This means that you may configure the resolution process in any way you like: change the precedence of searching through system properties and environment variables, or remove them entirely; add your own property sources to the mix as appropriate.

Concretely, the following statement works regardless of where the customer property is defined, as long as it is available in the Environment:

<beans>

<import resource="com/bank/service/${customer}-config.xml"/>

</beans>

1.14. Registering a LoadTimeWeaver

The LoadTimeWeaver is used by Spring to dynamically transform classes as they are loaded into the Java virtual machine (JVM).

To enable load-time weaving add the @EnableLoadTimeWeaving to one of your @Configuration classes:

@Configuration

@EnableLoadTimeWeaving

public class AppConfig {

}

Alternatively for XML configuration use the context:load-time-weaver element:

<beans>

<context:load-time-weaver/>

</beans>

Once configured for the ApplicationContext. Any bean within that ApplicationContext may implement LoadTimeWeaverAware, thereby receiving a reference to the load-time weaver instance. This is particularly useful in combination with Spring’s JPA support where load-time weaving may be necessary for JPA class transformation. Consult the LocalContainerEntityManagerFactoryBean javadocs for more detail. For more on AspectJ load-time weaving, see Load-time weaving with AspectJ in the Spring Framework.

1.15. Additional capabilities of the ApplicationContext

As was discussed in the chapter introduction, the org.springframework.beans.factory package provides basic functionality for managing and manipulating beans, including in a programmatic way. The org.springframework.context package adds the ApplicationContext interface, which extends the BeanFactory interface, in addition to extending other interfaces to provide additional functionality in a more application framework-oriented style. Many people use the ApplicationContext in a completely declarative fashion, not even creating it programmatically, but instead relying on support classes such as ContextLoader to automatically instantiate an ApplicationContext as part of the normal startup process of a Java EE web application.

To enhance BeanFactory functionality in a more framework-oriented style the context package also provides the following functionality:

Access to messages in i18n-style, through the MessageSource interface.

Access to resources, such as URLs and files, through the ResourceLoader interface.

Event publication to namely beans implementing the ApplicationListener interface, through the use of the ApplicationEventPublisher interface.

Loading of multiple (hierarchical) contexts, allowing each to be focused on one particular layer, such as the web layer of an application, through the HierarchicalBeanFactory interface.

1.15.1. Internationalization using MessageSource

The ApplicationContext interface extends an interface called MessageSource, and therefore provides internationalization (i18n) functionality. Spring also provides the interface HierarchicalMessageSource, which can resolve messages hierarchically. Together these interfaces provide the foundation upon which Spring effects message resolution. The methods defined on these interfaces include:

String getMessage(String code, Object[] args, String default, Locale loc): The basic method used to retrieve a message from the MessageSource. When no message is found for the specified locale, the default message is used. Any arguments passed in become replacement values, using the MessageFormat functionality provided by the standard library.

String getMessage(String code, Object[] args, Locale loc): Essentially the same as the previous method, but with one difference: no default message can be specified; if the message cannot be found, a NoSuchMessageException is thrown.

String getMessage(MessageSourceResolvable resolvable, Locale locale): All properties used in the preceding methods are also wrapped in a class named MessageSourceResolvable, which you can use with this method.

When an ApplicationContext is loaded, it automatically searches for a MessageSource bean defined in the context. The bean must have the name messageSource. If such a bean is found, all calls to the preceding methods are delegated to the message source. If no message source is found, the ApplicationContext attempts to find a parent containing a bean with the same name. If it does, it uses that bean as the MessageSource. If the ApplicationContext cannot find any source for messages, an empty DelegatingMessageSource is instantiated in order to be able to accept calls to the methods defined above.

Spring provides two MessageSource implementations, ResourceBundleMessageSource and StaticMessageSource. Both implement HierarchicalMessageSource in order to do nested messaging. The StaticMessageSource is rarely used but provides programmatic ways to add messages to the source. The ResourceBundleMessageSource is shown in the following example:

<beans>

<bean id="messageSource"

class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basenames">

<list>

<value>format</value>

<value>exceptions</value>

<value>windows</value>

</list>

</property>

</bean>

</beans>

In the example it is assumed you have three resource bundles defined in your classpath called format, exceptions and windows. Any request to resolve a message will be handled in the JDK standard way of resolving messages through ResourceBundles. For the purposes of the example, assume the contents of two of the above resource bundle files are…​

# in format.properties

message=Alligators rock!

# in exceptions.properties

argument.required=The {0} argument is required.

A program to execute the MessageSource functionality is shown in the next example. Remember that all ApplicationContext implementations are also MessageSource implementations and so can be cast to the MessageSource interface.

public static void main(String[] args) {

MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");

String message = resources.getMessage("message", null, "Default", null);

System.out.println(message);

}

The resulting output from the above program will be…​

Alligators rock!

So to summarize, the MessageSource is defined in a file called beans.xml, which exists at the root of your classpath. The messageSource bean definition refers to a number of resource bundles through its basenames property. The three files that are passed in the list to the basenames property exist as files at the root of your classpath and are called format.properties, exceptions.properties, and windows.properties respectively.

The next example shows arguments passed to the message lookup; these arguments will be converted into Strings and inserted into placeholders in the lookup message.

<beans>

<!-- this MessageSource is being used in a web application -->

<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basename" value="exceptions"/>

</bean>

<!-- lets inject the above MessageSource into this POJO -->

<bean id="example" class="com.foo.Example">

<property name="messages" ref="messageSource"/>

</bean>

</beans>

public class Example {

private MessageSource messages;

public void setMessages(MessageSource messages) {

this.messages = messages;

}

public void execute() {

String message = this.messages.getMessage("argument.required",

new Object [] {"userDao"}, "Required", null);

System.out.println(message);

}

}

The resulting output from the invocation of the execute() method will be…​

The userDao argument is required.

With regard to internationalization (i18n), Spring’s various MessageSource implementations follow the same locale resolution and fallback rules as the standard JDK ResourceBundle. In short, and continuing with the example messageSource defined previously, if you want to resolve messages against the British (en-GB) locale, you would create files called format\_en\_GB.properties, exceptions\_en\_GB.properties, and windows\_en\_GB.properties respectively.

Typically, locale resolution is managed by the surrounding environment of the application. In this example, the locale against which (British) messages will be resolved is specified manually.

# in exceptions\_en\_GB.properties

argument.required=Ebagum lad, the {0} argument is required, I say, required.

public static void main(final String[] args) {

MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");

String message = resources.getMessage("argument.required",

new Object [] {"userDao"}, "Required", Locale.UK);

System.out.println(message);

}

The resulting output from the running of the above program will be…​

Ebagum lad, the 'userDao' argument is required, I say, required.

You can also use the MessageSourceAware interface to acquire a reference to any MessageSource that has been defined. Any bean that is defined in an ApplicationContext that implements the MessageSourceAware interface is injected with the application context’s MessageSource when the bean is created and configured.

As an alternative to ResourceBundleMessageSource, Spring provides a ReloadableResourceBundleMessageSource class. This variant supports the same bundle file format but is more flexible than the standard JDK based ResourceBundleMessageSource implementation. In particular, it allows for reading files from any Spring resource location (not just from the classpath) and supports hot reloading of bundle property files (while efficiently caching them in between). Check out the ReloadableResourceBundleMessageSource javadocs for details.

1.15.2. Standard and custom events

Event handling in the ApplicationContext is provided through the ApplicationEvent class and ApplicationListener interface. If a bean that implements the ApplicationListener interface is deployed into the context, every time an ApplicationEvent gets published to the ApplicationContext, that bean is notified. Essentially, this is the standard Observer design pattern.

As of Spring 4.2, the event infrastructure has been significantly improved and offer an annotation-based model as well as the ability to publish any arbitrary event, that is an object that does not necessarily extend from ApplicationEvent. When such an object is published we wrap it in an event for you.

Spring provides the following standard events:

Table 7. Built-in Events

Event Explanation

ContextRefreshedEvent

Published when the ApplicationContext is initialized or refreshed, for example, using the refresh() method on the ConfigurableApplicationContext interface. "Initialized" here means that all beans are loaded, post-processor beans are detected and activated, singletons are pre-instantiated, and the ApplicationContext object is ready for use. As long as the context has not been closed, a refresh can be triggered multiple times, provided that the chosen ApplicationContext actually supports such "hot" refreshes. For example, XmlWebApplicationContext supports hot refreshes, but GenericApplicationContext does not.

ContextStartedEvent

Published when the ApplicationContext is started, using the start() method on the ConfigurableApplicationContext interface. "Started" here means that all Lifecycle beans receive an explicit start signal. Typically this signal is used to restart beans after an explicit stop, but it may also be used to start components that have not been configured for autostart , for example, components that have not already started on initialization.

ContextStoppedEvent

Published when the ApplicationContext is stopped, using the stop() method on the ConfigurableApplicationContext interface. "Stopped" here means that all Lifecycle beans receive an explicit stop signal. A stopped context may be restarted through a start() call.

ContextClosedEvent

Published when the ApplicationContext is closed, using the close() method on the ConfigurableApplicationContext interface. "Closed" here means that all singleton beans are destroyed. A closed context reaches its end of life; it cannot be refreshed or restarted.

RequestHandledEvent

A web-specific event telling all beans that an HTTP request has been serviced. This event is published after the request is complete. This event is only applicable to web applications using Spring’s DispatcherServlet.

You can also create and publish your own custom events. This example demonstrates a simple class that extends Spring’s ApplicationEvent base class:

public class BlackListEvent extends ApplicationEvent {

private final String address;

private final String test;

public BlackListEvent(Object source, String address, String test) {

super(source);

this.address = address;

this.test = test;

}

// accessor and other methods...

}

To publish a custom ApplicationEvent, call the publishEvent() method on an ApplicationEventPublisher. Typically this is done by creating a class that implements ApplicationEventPublisherAware and registering it as a Spring bean. The following example demonstrates such a class:

public class EmailService implements ApplicationEventPublisherAware {

private List<String> blackList;

private ApplicationEventPublisher publisher;

public void setBlackList(List<String> blackList) {

this.blackList = blackList;

}

public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {

this.publisher = publisher;

}

public void sendEmail(String address, String text) {

if (blackList.contains(address)) {

BlackListEvent event = new BlackListEvent(this, address, text);

publisher.publishEvent(event);

return;

}

// send email...

}

}

At configuration time, the Spring container will detect that EmailService implements ApplicationEventPublisherAware and will automatically call setApplicationEventPublisher(). In reality, the parameter passed in will be the Spring container itself; you’re simply interacting with the application context via its ApplicationEventPublisher interface.

To receive the custom ApplicationEvent, create a class that implements ApplicationListener and register it as a Spring bean. The following example demonstrates such a class:

public class BlackListNotifier implements ApplicationListener<BlackListEvent> {

private String notificationAddress;

public void setNotificationAddress(String notificationAddress) {

this.notificationAddress = notificationAddress;

}

public void onApplicationEvent(BlackListEvent event) {

// notify appropriate parties via notificationAddress...

}

}

Notice that ApplicationListener is generically parameterized with the type of your custom event, BlackListEvent. This means that the onApplicationEvent() method can remain type-safe, avoiding any need for downcasting. You may register as many event listeners as you wish, but note that by default event listeners receive events synchronously. This means the publishEvent() method blocks until all listeners have finished processing the event. One advantage of this synchronous and single-threaded approach is that when a listener receives an event, it operates inside the transaction context of the publisher if a transaction context is available. If another strategy for event publication becomes necessary, refer to the javadoc for Spring’s ApplicationEventMulticaster interface.

The following example shows the bean definitions used to register and configure each of the classes above:

<bean id="emailService" class="example.EmailService">

<property name="blackList">

<list>

<value>known.spammer@example.org</value>

<value>known.hacker@example.org</value>

<value>john.doe@example.org</value>

</list>

</property>

</bean>

<bean id="blackListNotifier" class="example.BlackListNotifier">

<property name="notificationAddress" value="blacklist@example.org"/>

</bean>

Putting it all together, when the sendEmail() method of the emailService bean is called, if there are any emails that should be blacklisted, a custom event of type BlackListEvent is published. The blackListNotifier bean is registered as an ApplicationListener and thus receives the BlackListEvent, at which point it can notify appropriate parties.

Spring’s eventing mechanism is designed for simple communication between Spring beans within the same application context. However, for more sophisticated enterprise integration needs, the separately-maintained Spring Integration project provides complete support for building lightweight, pattern-oriented, event-driven architectures that build upon the well-known Spring programming model.

Annotation-based event listeners

As of Spring 4.2, an event listener can be registered on any public method of a managed bean via the EventListener annotation. The BlackListNotifier can be rewritten as follows:

public class BlackListNotifier {

private String notificationAddress;

public void setNotificationAddress(String notificationAddress) {

this.notificationAddress = notificationAddress;

}

@EventListener

public void processBlackListEvent(BlackListEvent event) {

// notify appropriate parties via notificationAddress...

}

}

As you can see above, the method signature once again declares the event type it listens to, but this time with a flexible name and without implementing a specific listener interface. The event type can also be narrowed through generics as long as the actual event type resolves your generic parameter in its implementation hierarchy.

If your method should listen to several events or if you want to define it with no parameter at all, the event type(s) can also be specified on the annotation itself:

@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})

public void handleContextStart() {

...

}

It is also possible to add additional runtime filtering via the condition attribute of the annotation that defines a SpEL expression that should match to actually invoke the method for a particular event.

For instance, our notifier can be rewritten to be only invoked if the test attribute of the event is equal to foo:

@EventListener(condition = "#blEvent.test == 'foo'")

public void processBlackListEvent(BlackListEvent blEvent) {

// notify appropriate parties via notificationAddress...

}

Each SpEL expression evaluates again a dedicated context. The next table lists the items made available to the context so one can use them for conditional event processing:

Table 8. Event SpEL available metadata

Name Location Description Example

Event

root object

The actual ApplicationEvent

#root.event

Arguments array

root object

The arguments (as array) used for invoking the target

#root.args[0]

Argument name

evaluation context

Name of any of the method arguments. If for some reason the names are not available (e.g. no debug information), the argument names are also available under the #a<#arg> where #arg stands for the argument index (starting from 0).

#blEvent or #a0 (one can also use #p0 or #p<#arg> notation as an alias).

Note that #root.event allows you to access to the underlying event, even if your method signature actually refers to an arbitrary object that was published.

If you need to publish an event as the result of processing another, just change the method signature to return the event that should be published, something like:

@EventListener

public ListUpdateEvent handleBlackListEvent(BlackListEvent event) {

// notify appropriate parties via notificationAddress and

// then publish a ListUpdateEvent...

}

This feature is not supported for asynchronous listeners.

This new method will publish a new ListUpdateEvent for every BlackListEvent handled by the method above. If you need to publish several events, just return a Collection of events instead.

Asynchronous Listeners

If you want a particular listener to process events asynchronously, simply reuse the regular @Async support:

@EventListener

@Async

public void processBlackListEvent(BlackListEvent event) {

// BlackListEvent is processed in a separate thread

}

Be aware of the following limitations when using asynchronous events:

If the event listener throws an Exception it will not be propagated to the caller, check AsyncUncaughtExceptionHandler for more details.

Such event listener cannot send replies. If you need to send another event as the result of the processing, inject ApplicationEventPublisher to send the event manually.

Ordering listeners

If you need the listener to be invoked before another one, just add the @Order annotation to the method declaration:

@EventListener

@Order(42)

public void processBlackListEvent(BlackListEvent event) {

// notify appropriate parties via notificationAddress...

}

Generic events

You may also use generics to further define the structure of your event. Consider an EntityCreatedEvent<T> where T is the type of the actual entity that got created. You can create the following listener definition to only receive EntityCreatedEvent for a Person:

@EventListener

public void onPersonCreated(EntityCreatedEvent<Person> event) {

...

}

Due to type erasure, this will only work if the event that is fired resolves the generic parameter(s) on which the event listener filters on (that is something like class PersonCreatedEvent extends EntityCreatedEvent<Person> { …​ }).

In certain circumstances, this may become quite tedious if all events follow the same structure (as it should be the case for the event above). In such a case, you can implement ResolvableTypeProvider to guide the framework beyond what the runtime environment provides:

public class EntityCreatedEvent<T>

extends ApplicationEvent implements ResolvableTypeProvider {

public EntityCreatedEvent(T entity) {

super(entity);

}

@Override

public ResolvableType getResolvableType() {

return ResolvableType.forClassWithGenerics(getClass(),

ResolvableType.forInstance(getSource()));

}

}

This works not only for ApplicationEvent but any arbitrary object that you’d send as an event.

1.15.3. Convenient access to low-level resources

For optimal usage and understanding of application contexts, users should generally familiarize themselves with Spring’s Resource abstraction, as described in the chapter Resources.

An application context is a ResourceLoader, which can be used to load Resources. A Resource is essentially a more feature rich version of the JDK class java.net.URL, in fact, the implementations of the Resource wrap an instance of java.net.URL where appropriate. A Resource can obtain low-level resources from almost any location in a transparent fashion, including from the classpath, a filesystem location, anywhere describable with a standard URL, and some other variations. If the resource location string is a simple path without any special prefixes, where those resources come from is specific and appropriate to the actual application context type.

You can configure a bean deployed into the application context to implement the special callback interface, ResourceLoaderAware, to be automatically called back at initialization time with the application context itself passed in as the ResourceLoader. You can also expose properties of type Resource, to be used to access static resources; they will be injected into it like any other properties. You can specify those Resource properties as simple String paths, and rely on a special JavaBean PropertyEditor that is automatically registered by the context, to convert those text strings to actual Resource objects when the bean is deployed.

The location path or paths supplied to an ApplicationContext constructor are actually resource strings, and in simple form are treated appropriately to the specific context implementation. ClassPathXmlApplicationContext treats a simple location path as a classpath location. You can also use location paths (resource strings) with special prefixes to force loading of definitions from the classpath or a URL, regardless of the actual context type.

1.15.4. Convenient ApplicationContext instantiation for web applications

You can create ApplicationContext instances declaratively by using, for example, a ContextLoader. Of course you can also create ApplicationContext instances programmatically by using one of the ApplicationContext implementations.

You can register an ApplicationContext using the ContextLoaderListener as follows:

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

The listener inspects the contextConfigLocation parameter. If the parameter does not exist, the listener uses /WEB-INF/applicationContext.xml as a default. When the parameter does exist, the listener separates the String by using predefined delimiters (comma, semicolon and whitespace) and uses the values as locations where application contexts will be searched. Ant-style path patterns are supported as well. Examples are /WEB-INF/\*Context.xml for all files with names ending with "Context.xml", residing in the "WEB-INF" directory, and /WEB-INF/\*\*/\*Context.xml, for all such files in any subdirectory of "WEB-INF".

1.15.5. Deploying a Spring ApplicationContext as a Java EE RAR file

It is possible to deploy a Spring ApplicationContext as a RAR file, encapsulating the context and all of its required bean classes and library JARs in a Java EE RAR deployment unit. This is the equivalent of bootstrapping a standalone ApplicationContext, just hosted in Java EE environment, being able to access the Java EE servers facilities. RAR deployment is more natural alternative to scenario of deploying a headless WAR file, in effect, a WAR file without any HTTP entry points that is used only for bootstrapping a Spring ApplicationContext in a Java EE environment.

RAR deployment is ideal for application contexts that do not need HTTP entry points but rather consist only of message endpoints and scheduled jobs. Beans in such a context can use application server resources such as the JTA transaction manager and JNDI-bound JDBC DataSources and JMS ConnectionFactory instances, and may also register with the platform’s JMX server - all through Spring’s standard transaction management and JNDI and JMX support facilities. Application components can also interact with the application server’s JCA WorkManager through Spring’s TaskExecutor abstraction.

Check out the javadoc of the SpringContextResourceAdapter class for the configuration details involved in RAR deployment.

For a simple deployment of a Spring ApplicationContext as a Java EE RAR file: package all application classes into a RAR file, which is a standard JAR file with a different file extension. Add all required library JARs into the root of the RAR archive. Add a "META-INF/ra.xml" deployment descriptor (as shown in SpringContextResourceAdapters javadoc) and the corresponding Spring XML bean definition file(s) (typically "META-INF/applicationContext.xml"), and drop the resulting RAR file into your application server’s deployment directory.

Such RAR deployment units are usually self-contained; they do not expose components to the outside world, not even to other modules of the same application. Interaction with a RAR-based ApplicationContext usually occurs through JMS destinations that it shares with other modules. A RAR-based ApplicationContext may also, for example, schedule some jobs, reacting to new files in the file system (or the like). If it needs to allow synchronous access from the outside, it could for example export RMI endpoints, which of course may be used by other application modules on the same machine.

1.16. The BeanFactory

The BeanFactory provides the underlying basis for Spring’s IoC functionality but it is only used directly in integration with other third-party frameworks and is now largely historical in nature for most users of Spring. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring. Often third-party components that can not use more modern equivalents such as @PostConstruct or @PreDestroy in order to avoid a dependency on JSR-250.

This section provides additional background into the differences between the BeanFactory and ApplicationContext and how one might access the IoC container directly through a classic singleton lookup.

1.16.1. BeanFactory or ApplicationContext?

Use an ApplicationContext unless you have a good reason for not doing so.

Because the ApplicationContext includes all functionality of the BeanFactory, it is generally recommended over the BeanFactory, except for a few situations such as in embedded applications running on resource-constrained devices where memory consumption might be critical and a few extra kilobytes might make a difference. However, for most typical enterprise applications and systems, the ApplicationContext is what you will want to use. Spring makes heavy use of the BeanPostProcessor extension point (to effect proxying and so on). If you use only a plain BeanFactory, a fair amount of support such as transactions and AOP will not take effect, at least not without some extra steps on your part. This situation could be confusing because nothing is actually wrong with the configuration.

The following table lists features provided by the BeanFactory and ApplicationContext interfaces and implementations.

Table 9. Feature Matrix

Feature BeanFactory ApplicationContext

Bean instantiation/wiring

Yes

Yes

Automatic BeanPostProcessor registration

No

Yes

Automatic BeanFactoryPostProcessor registration

No

Yes

Convenient MessageSource access (for i18n)

No

Yes

ApplicationEvent publication

No

Yes

To explicitly register a bean post-processor with a BeanFactory implementation, you need to write code like this:

DefaultListableBeanFactory factory = new DefaultListableBeanFactory();

// populate the factory with bean definitions

// now register any needed BeanPostProcessor instances

MyBeanPostProcessor postProcessor = new MyBeanPostProcessor();

factory.addBeanPostProcessor(postProcessor);

// now start using the factory

To explicitly register a BeanFactoryPostProcessor when using a BeanFactory implementation, you must write code like this:

DefaultListableBeanFactory factory = new DefaultListableBeanFactory();

XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);

reader.loadBeanDefinitions(new FileSystemResource("beans.xml"));

// bring in some property values from a Properties file

PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();

cfg.setLocation(new FileSystemResource("jdbc.properties"));

// now actually do the replacement

cfg.postProcessBeanFactory(factory);

In both cases, the explicit registration step is inconvenient, which is one reason why the various ApplicationContext implementations are preferred above plain BeanFactory implementations in the vast majority of Spring-backed applications, especially when using BeanFactoryPostProcessors and BeanPostProcessors. These mechanisms implement important functionality such as property placeholder replacement and AOP.

2. Resources

2.1. 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.

2.2. 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.

2.3. Built-in Resource implementations

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

2.3.1. 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.

2.3.2. 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.

2.3.3. FileSystemResource

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

2.3.4. 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.

2.3.5. 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.

2.3.6. 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.

2.4. 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 Strings to Resources:

Table 10. 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. [3]

http:

http://myserver/logo.png

Loaded as a URL.

(none)

/data/config.xml

Depends on the underlying ApplicationContext.

2.5. 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 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 @Autowired.

2.6. 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"/>

2.7. Application contexts and Resource paths

2.7.1. 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 ClassPathXmlApplicationContext javadocs for details on the various constructors.

2.7.2. 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 might not retrieve files from the root of jar files but rather only from the root of expanded directories.

Spring’s ability to retrieve classpath entries originates from the JDK’s ClassLoader.getResources() method which only returns file system locations for a passed-in empty string (indicating potential roots to search). Spring evaluates URLClassLoader runtime configuration and the "java.class.path" manifest in jar files as well but this is not guaranteed to lead to portable behavior.

The scanning of classpath packages requires the presence of corresponding directory entries in the classpath. When you build JARs with Ant, make sure that you do not activate the files-only switch of the JAR task. Also, classpath directories may not get exposed based on security policies in some environments, e.g. standalone apps on JDK 1.7.0\_45 and higher (which requires 'Trusted-Library' setup in your manifests; see http://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources).

On JDK 9’s module path (Jigsaw), Spring’s classpath scanning generally works as expected. Putting resources into a dedicated directory is highly recommendable here as well, avoiding the aforementioned portability problems with searching the jar file root level.

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.

2.7.3. 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");

3. Validation, Data Binding, and Type Conversion

3.1. Introduction

JSR-303/JSR-349 Bean Validation

Spring Framework 4.0 supports Bean Validation 1.0 (JSR-303) and Bean Validation 1.1 (JSR-349) in terms of setup support, also adapting it to Spring’s Validator interface.

An application can choose to enable Bean Validation once globally, as described in Spring Validation, and use it exclusively for all validation needs.

An application can also register additional Spring Validator instances per DataBinder instance, as described in Configuring a DataBinder. This may be useful for plugging in validation logic without the use of annotations.

There are pros and cons for considering validation as business logic, and Spring offers a design for validation (and data binding) that does not exclude either one of them. Specifically validation should not be tied to the web tier, should be easy to localize and it should be possible to plug in any validator available. Considering the above, Spring has come up with a Validator interface that is both basic and eminently usable in every layer of an application.

Data binding is useful for allowing user input to be dynamically bound to the domain model of an application (or whatever objects you use to process user input). Spring provides the so-called DataBinder to do exactly that. The Validator and the DataBinder make up the validation package, which is primarily used in but not limited to the MVC framework.

The BeanWrapper is a fundamental concept in the Spring Framework and is used in a lot of places. However, you probably will not have the need to use the BeanWrapper directly. Because this is reference documentation however, we felt that some explanation might be in order. We will explain the BeanWrapper in this chapter since, if you were going to use it at all, you would most likely do so when trying to bind data to objects.

Spring’s DataBinder and the lower-level BeanWrapper both use PropertyEditors to parse and format property values. The PropertyEditor concept is part of the JavaBeans specification, and is also explained in this chapter. Spring 3 introduces a "core.convert" package that provides a general type conversion facility, as well as a higher-level "format" package for formatting UI field values. These new packages may be used as simpler alternatives to PropertyEditors, and will also be discussed in this chapter.

3.2. Validation using Spring’s Validator interface

Spring features a Validator interface that you can use to validate objects. The Validator interface works using an Errors object so that while validating, validators can report validation failures to the Errors object.

Let’s consider a small data object:

public class Person {

private String name;

private int age;

// the usual getters and setters...

}

We’re going to provide validation behavior for the Person class by implementing the following two methods of the org.springframework.validation.Validator interface:

supports(Class) - Can this Validator validate instances of the supplied Class?

validate(Object, org.springframework.validation.Errors) - validates the given object and in case of validation errors, registers those with the given Errors object

Implementing a Validator is fairly straightforward, especially when you know of the ValidationUtils helper class that the Spring Framework also provides.

public class PersonValidator implements Validator {

/\*\*

\* This Validator validates \*just\* Person instances

\*/

public boolean supports(Class clazz) {

return Person.class.equals(clazz);

}

public void validate(Object obj, Errors e) {

ValidationUtils.rejectIfEmpty(e, "name", "name.empty");

Person p = (Person) obj;

if (p.getAge() < 0) {

e.rejectValue("age", "negativevalue");

} else if (p.getAge() > 110) {

e.rejectValue("age", "too.darn.old");

}

}

}

As you can see, the static rejectIfEmpty(..) method on the ValidationUtils class is used to reject the 'name' property if it is null or the empty string. Have a look at the ValidationUtils javadocs to see what functionality it provides besides the example shown previously.

While it is certainly possible to implement a single Validator class to validate each of the nested objects in a rich object, it may be better to encapsulate the validation logic for each nested class of object in its own Validator implementation. A simple example of a 'rich' object would be a Customer that is composed of two String properties (a first and second name) and a complex Address object. Address objects may be used independently of Customer objects, and so a distinct AddressValidator has been implemented. If you want your CustomerValidator to reuse the logic contained within the AddressValidator class without resorting to copy-and-paste, you can dependency-inject or instantiate an AddressValidator within your CustomerValidator, and use it like so:

public class CustomerValidator implements Validator {

private final Validator addressValidator;

public CustomerValidator(Validator addressValidator) {

if (addressValidator == null) {

throw new IllegalArgumentException("The supplied [Validator] is " +

"required and must not be null.");

}

if (!addressValidator.supports(Address.class)) {

throw new IllegalArgumentException("The supplied [Validator] must " +

"support the validation of [Address] instances.");

}

this.addressValidator = addressValidator;

}

/\*\*

\* This Validator validates Customer instances, and any subclasses of Customer too

\*/

public boolean supports(Class clazz) {

return Customer.class.isAssignableFrom(clazz);

}

public void validate(Object target, Errors errors) {

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "field.required");

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required");

Customer customer = (Customer) target;

try {

errors.pushNestedPath("address");

ValidationUtils.invokeValidator(this.addressValidator, customer.getAddress(), errors);

} finally {

errors.popNestedPath();

}

}

}

Validation errors are reported to the Errors object passed to the validator. In case of Spring Web MVC you can use <spring:bind/> tag to inspect the error messages, but of course you can also inspect the errors object yourself. More information about the methods it offers can be found in the javadocs.

3.3. Resolving codes to error messages

We’ve talked about databinding and validation. Outputting messages corresponding to validation errors is the last thing we need to discuss. In the example we’ve shown above, we rejected the name and the age field. If we’re going to output the error messages by using a MessageSource, we will do so using the error code we’ve given when rejecting the field ('name' and 'age' in this case). When you call (either directly, or indirectly, using for example the ValidationUtils class) rejectValue or one of the other reject methods from the Errors interface, the underlying implementation will not only register the code you’ve passed in, but also a number of additional error codes. What error codes it registers is determined by the MessageCodesResolver that is used. By default, the DefaultMessageCodesResolver is used, which for example not only registers a message with the code you gave, but also messages that include the field name you passed to the reject method. So in case you reject a field using rejectValue("age", "too.darn.old"), apart from the too.darn.old code, Spring will also register too.darn.old.age and too.darn.old.age.int (so the first will include the field name and the second will include the type of the field); this is done as a convenience to aid developers in targeting error messages and suchlike.

More information on the MessageCodesResolver and the default strategy can be found online in the javadocs of MessageCodesResolver and DefaultMessageCodesResolver, respectively.

3.4. Bean manipulation and the BeanWrapper

The org.springframework.beans package adheres to the JavaBeans standard provided by Oracle. A JavaBean is simply a class with a default no-argument constructor, which follows a naming convention where (by way of an example) a property named bingoMadness would have a setter method setBingoMadness(..) and a getter method getBingoMadness(). For more information about JavaBeans and the specification, please refer to Oracle’s website ( javabeans).

One quite important class in the beans package is the BeanWrapper interface and its corresponding implementation ( BeanWrapperImpl). As quoted from the javadocs, the BeanWrapper offers functionality to set and get property values (individually or in bulk), get property descriptors, and to query properties to determine if they are readable or writable. Also, the BeanWrapper offers support for nested properties, enabling the setting of properties on sub-properties to an unlimited depth. Then, the BeanWrapper supports the ability to add standard JavaBeans PropertyChangeListeners and VetoableChangeListeners, without the need for supporting code in the target class. Last but not least, the BeanWrapper provides support for the setting of indexed properties. The BeanWrapper usually isn’t used by application code directly, but by the DataBinder and the BeanFactory.

The way the BeanWrapper works is partly indicated by its name: it wraps a bean to perform actions on that bean, like setting and retrieving properties.

3.4.1. Setting and getting basic and nested properties

Setting and getting properties is done using the setPropertyValue(s) and getPropertyValue(s) methods that both come with a couple of overloaded variants. They’re all described in more detail in the javadocs Spring comes with. What’s important to know is that there are a couple of conventions for indicating properties of an object. A couple of examples:

Table 11. Examples of properties

Expression Explanation

name

Indicates the property name corresponding to the methods getName() or isName() and setName(..)

account.name

Indicates the nested property name of the property account corresponding e.g. to the methods getAccount().setName() or getAccount().getName()

account[2]

Indicates the third element of the indexed property account. Indexed properties can be of type array, list or other naturally ordered collection

account[COMPANYNAME]

Indicates the value of the map entry indexed by the key COMPANYNAME of the Map property account

Below you’ll find some examples of working with the BeanWrapper to get and set properties.

(This next section is not vitally important to you if you’re not planning to work with the BeanWrapper directly. If you’re just using the DataBinder and the BeanFactory and their out-of-the-box implementation, you should skip ahead to the section about PropertyEditors.)

Consider the following two classes:

public class Company {

private String name;

private Employee managingDirector;

public String getName() {

return this.name;

}

public void setName(String name) {

this.name = name;

}

public Employee getManagingDirector() {

return this.managingDirector;

}

public void setManagingDirector(Employee managingDirector) {

this.managingDirector = managingDirector;

}

}

public class Employee {

private String name;

private float salary;

public String getName() {

return this.name;

}

public void setName(String name) {

this.name = name;

}

public float getSalary() {

return salary;

}

public void setSalary(float salary) {

this.salary = salary;

}

}

The following code snippets show some examples of how to retrieve and manipulate some of the properties of instantiated Companies and Employees:

BeanWrapper company = new BeanWrapperImpl(new Company());

// setting the company name..

company.setPropertyValue("name", "Some Company Inc.");

// ... can also be done like this:

PropertyValue value = new PropertyValue("name", "Some Company Inc.");

company.setPropertyValue(value);

// ok, let's create the director and tie it to the company:

BeanWrapper jim = new BeanWrapperImpl(new Employee());

jim.setPropertyValue("name", "Jim Stravinsky");

company.setPropertyValue("managingDirector", jim.getWrappedInstance());

// retrieving the salary of the managingDirector through the company

Float salary = (Float) company.getPropertyValue("managingDirector.salary");

3.4.2. Built-in PropertyEditor implementations

Spring uses the concept of PropertyEditors to effect the conversion between an Object and a String. If you think about it, it sometimes might be handy to be able to represent properties in a different way than the object itself. For example, a Date can be represented in a human readable way (as the String '2007-14-09'), while we’re still able to convert the human readable form back to the original date (or even better: convert any date entered in a human readable form, back to Date objects). This behavior can be achieved by registering custom editors, of type java.beans.PropertyEditor. Registering custom editors on a BeanWrapper or alternately in a specific IoC container as mentioned in the previous chapter, gives it the knowledge of how to convert properties to the desired type. Read more about PropertyEditors in the javadocs of the java.beans package provided by Oracle.

A couple of examples where property editing is used in Spring:

setting properties on beans is done using PropertyEditors. When mentioning java.lang.String as the value of a property of some bean you’re declaring in XML file, Spring will (if the setter of the corresponding property has a Class-parameter) use the ClassEditor to try to resolve the parameter to a Class object.

parsing HTTP request parameters in Spring’s MVC framework is done using all kinds of PropertyEditors that you can manually bind in all subclasses of the CommandController.

Spring has a number of built-in PropertyEditors to make life easy. Each of those is listed below and they are all located in the org.springframework.beans.propertyeditors package. Most, but not all (as indicated below), are registered by default by BeanWrapperImpl. Where the property editor is configurable in some fashion, you can of course still register your own variant to override the default one:

Table 12. Built-in PropertyEditors

Class Explanation

ByteArrayPropertyEditor

Editor for byte arrays. Strings will simply be converted to their corresponding byte representations. Registered by default by BeanWrapperImpl.

ClassEditor

Parses Strings representing classes to actual classes and the other way around. When a class is not found, an IllegalArgumentException is thrown. Registered by default by BeanWrapperImpl.

CustomBooleanEditor

Customizable property editor for Boolean properties. Registered by default by BeanWrapperImpl, but, can be overridden by registering custom instance of it as custom editor.

CustomCollectionEditor

Property editor for Collections, converting any source Collection to a given target Collection type.

CustomDateEditor

Customizable property editor for java.util.Date, supporting a custom DateFormat. NOT registered by default. Must be user registered as needed with appropriate format.

CustomNumberEditor

Customizable property editor for any Number subclass like Integer, Long, Float, Double. Registered by default by BeanWrapperImpl, but can be overridden by registering custom instance of it as a custom editor.

FileEditor

Capable of resolving Strings to java.io.File objects. Registered by default by BeanWrapperImpl.

InputStreamEditor

One-way property editor, capable of taking a text string and producing (via an intermediate ResourceEditor and Resource) an InputStream, so InputStream properties may be directly set as Strings. Note that the default usage will not close the InputStream for you! Registered by default by BeanWrapperImpl.

LocaleEditor

Capable of resolving Strings to Locale objects and vice versa (the String format is [country][variant], which is the same thing the toString() method of Locale provides). Registered by default by BeanWrapperImpl.

PatternEditor

Capable of resolving Strings to java.util.regex.Pattern objects and vice versa.

PropertiesEditor

Capable of converting Strings (formatted using the format as defined in the javadocs of the java.util.Properties class) to Properties objects. Registered by default by BeanWrapperImpl.

StringTrimmerEditor

Property editor that trims Strings. Optionally allows transforming an empty string into a null value. NOT registered by default; must be user registered as needed.

URLEditor

Capable of resolving a String representation of a URL to an actual URL object. Registered by default by BeanWrapperImpl.

Spring uses the java.beans.PropertyEditorManager to set the search path for property editors that might be needed. The search path also includes sun.bean.editors, which includes PropertyEditor implementations for types such as Font, Color, and most of the primitive types. Note also that the standard JavaBeans infrastructure will automatically discover PropertyEditor classes (without you having to register them explicitly) if they are in the same package as the class they handle, and have the same name as that class, with 'Editor' appended; for example, one could have the following class and package structure, which would be sufficient for the FooEditor class to be recognized and used as the PropertyEditor for Foo-typed properties.

com

chank

pop

Foo

FooEditor // the PropertyEditor for the Foo class

Note that you can also use the standard BeanInfo JavaBeans mechanism here as well (described in not-amazing-detail here). Find below an example of using the BeanInfo mechanism for explicitly registering one or more PropertyEditor instances with the properties of an associated class.

com

chank

pop

Foo

FooBeanInfo // the BeanInfo for the Foo class

Here is the Java source code for the referenced FooBeanInfo class. This would associate a CustomNumberEditor with the age property of the Foo class.

public class FooBeanInfo extends SimpleBeanInfo {

public PropertyDescriptor[] getPropertyDescriptors() {

try {

final PropertyEditor numberPE = new CustomNumberEditor(Integer.class, true);

PropertyDescriptor ageDescriptor = new PropertyDescriptor("age", Foo.class) {

public PropertyEditor createPropertyEditor(Object bean) {

return numberPE;

};

};

return new PropertyDescriptor[] { ageDescriptor };

}

catch (IntrospectionException ex) {

throw new Error(ex.toString());

}

}

}

Registering additional custom PropertyEditors

When setting bean properties as a string value, a Spring IoC container ultimately uses standard JavaBeans PropertyEditors to convert these Strings to the complex type of the property. Spring pre-registers a number of custom PropertyEditors (for example, to convert a classname expressed as a string into a real Class object). Additionally, Java’s standard JavaBeans PropertyEditor lookup mechanism allows a PropertyEditor for a class simply to be named appropriately and placed in the same package as the class it provides support for, to be found automatically.

If there is a need to register other custom PropertyEditors, there are several mechanisms available. The most manual approach, which is not normally convenient or recommended, is to simply use the registerCustomEditor() method of the ConfigurableBeanFactory interface, assuming you have a BeanFactory reference. Another, slightly more convenient, mechanism is to use a special bean factory post-processor called CustomEditorConfigurer. Although bean factory post-processors can be used with BeanFactory implementations, the CustomEditorConfigurer has a nested property setup, so it is strongly recommended that it is used with the ApplicationContext, where it may be deployed in similar fashion to any other bean, and automatically detected and applied.

Note that all bean factories and application contexts automatically use a number of built-in property editors, through their use of something called a BeanWrapper to handle property conversions. The standard property editors that the BeanWrapper registers are listed in the previous section. Additionally, ApplicationContexts also override or add an additional number of editors to handle resource lookups in a manner appropriate to the specific application context type.

Standard JavaBeans PropertyEditor instances are used to convert property values expressed as strings to the actual complex type of the property. CustomEditorConfigurer, a bean factory post-processor, may be used to conveniently add support for additional PropertyEditor instances to an ApplicationContext.

Consider a user class ExoticType, and another class DependsOnExoticType which needs ExoticType set as a property:

package example;

public class ExoticType {

private String name;

public ExoticType(String name) {

this.name = name;

}

}

public class DependsOnExoticType {

private ExoticType type;

public void setType(ExoticType type) {

this.type = type;

}

}

When things are properly set up, we want to be able to assign the type property as a string, which a PropertyEditor will behind the scenes convert into an actual ExoticType instance:

<bean id="sample" class="example.DependsOnExoticType">

<property name="type" value="aNameForExoticType"/>

</bean>

The PropertyEditor implementation could look similar to this:

// converts string representation to ExoticType object

package example;

public class ExoticTypeEditor extends PropertyEditorSupport {

public void setAsText(String text) {

setValue(new ExoticType(text.toUpperCase()));

}

}

Finally, we use CustomEditorConfigurer to register the new PropertyEditor with the ApplicationContext, which will then be able to use it as needed:

<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">

<property name="customEditors">

<map>

<entry key="example.ExoticType" value="example.ExoticTypeEditor"/>

</map>

</property>

</bean>

Using PropertyEditorRegistrars

Another mechanism for registering property editors with the Spring container is to create and use a PropertyEditorRegistrar. This interface is particularly useful when you need to use the same set of property editors in several different situations: write a corresponding registrar and reuse that in each case. PropertyEditorRegistrars work in conjunction with an interface called PropertyEditorRegistry, an interface that is implemented by the Spring BeanWrapper (and DataBinder). PropertyEditorRegistrars are particularly convenient when used in conjunction with the CustomEditorConfigurer (introduced here), which exposes a property called setPropertyEditorRegistrars(..): PropertyEditorRegistrars added to a CustomEditorConfigurer in this fashion can easily be shared with DataBinder and Spring MVC Controllers. Furthermore, it avoids the need for synchronization on custom editors: a PropertyEditorRegistrar is expected to create fresh PropertyEditor instances for each bean creation attempt.

Using a PropertyEditorRegistrar is perhaps best illustrated with an example. First off, you need to create your own PropertyEditorRegistrar implementation:

package com.foo.editors.spring;

public final class CustomPropertyEditorRegistrar implements PropertyEditorRegistrar {

public void registerCustomEditors(PropertyEditorRegistry registry) {

// it is expected that new PropertyEditor instances are created

registry.registerCustomEditor(ExoticType.class, new ExoticTypeEditor());

// you could register as many custom property editors as are required here...

}

}

See also the org.springframework.beans.support.ResourceEditorRegistrar for an example PropertyEditorRegistrar implementation. Notice how in its implementation of the registerCustomEditors(..) method it creates new instances of each property editor.

Next we configure a CustomEditorConfigurer and inject an instance of our CustomPropertyEditorRegistrar into it:

<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">

<property name="propertyEditorRegistrars">

<list>

<ref bean="customPropertyEditorRegistrar"/>

</list>

</property>

</bean>

<bean id="customPropertyEditorRegistrar"

class="com.foo.editors.spring.CustomPropertyEditorRegistrar"/>

Finally, and in a bit of a departure from the focus of this chapter, for those of you using Spring’s MVC web framework, using PropertyEditorRegistrars in conjunction with data-binding Controllers (such as SimpleFormController) can be very convenient. Find below an example of using a PropertyEditorRegistrar in the implementation of an initBinder(..) method:

public final class RegisterUserController extends SimpleFormController {

private final PropertyEditorRegistrar customPropertyEditorRegistrar;

public RegisterUserController(PropertyEditorRegistrar propertyEditorRegistrar) {

this.customPropertyEditorRegistrar = propertyEditorRegistrar;

}

protected void initBinder(HttpServletRequest request,

ServletRequestDataBinder binder) throws Exception {

this.customPropertyEditorRegistrar.registerCustomEditors(binder);

}

// other methods to do with registering a User

}

This style of PropertyEditor registration can lead to concise code (the implementation of initBinder(..) is just one line long!), and allows common PropertyEditor registration code to be encapsulated in a class and then shared amongst as many Controllers as needed.

3.5. Spring Type Conversion

Spring 3 introduces a core.convert package that provides a general type conversion system. The system defines an SPI to implement type conversion logic, as well as an API to execute type conversions at runtime. Within a Spring container, this system can be used as an alternative to PropertyEditors to convert externalized bean property value strings to required property types. The public API may also be used anywhere in your application where type conversion is needed.

3.5.1. Converter SPI

The SPI to implement type conversion logic is simple and strongly typed:

package org.springframework.core.convert.converter;

public interface Converter<S, T> {

T convert(S source);

}

To create your own converter, simply implement the interface above. Parameterize S as the type you are converting from, and T as the type you are converting to. Such a converter can also be applied transparently if a collection or array of S needs to be converted to an array or collection of T, provided that a delegating array/collection converter has been registered as well (which DefaultConversionService does by default).

For each call to convert(S), the source argument is guaranteed to be NOT null. Your Converter may throw any unchecked exception if conversion fails; specifically, an IllegalArgumentException should be thrown to report an invalid source value. Take care to ensure that your Converter implementation is thread-safe.

Several converter implementations are provided in the core.convert.support package as a convenience. These include converters from Strings to Numbers and other common types. Consider StringToInteger as an example for a typical Converter implementation:

package org.springframework.core.convert.support;

final class StringToInteger implements Converter<String, Integer> {

public Integer convert(String source) {

return Integer.valueOf(source);

}

}

3.5.2. ConverterFactory

When you need to centralize the conversion logic for an entire class hierarchy, for example, when converting from String to java.lang.Enum objects, implement ConverterFactory:

package org.springframework.core.convert.converter;

public interface ConverterFactory<S, R> {

<T extends R> Converter<S, T> getConverter(Class<T> targetType);

}

Parameterize S to be the type you are converting from and R to be the base type defining the range of classes you can convert to. Then implement getConverter(Class<T>), where T is a subclass of R.

Consider the StringToEnum ConverterFactory as an example:

package org.springframework.core.convert.support;

final class StringToEnumConverterFactory implements ConverterFactory<String, Enum> {

public <T extends Enum> Converter<String, T> getConverter(Class<T> targetType) {

return new StringToEnumConverter(targetType);

}

private final class StringToEnumConverter<T extends Enum> implements Converter<String, T> {

private Class<T> enumType;

public StringToEnumConverter(Class<T> enumType) {

this.enumType = enumType;

}

public T convert(String source) {

return (T) Enum.valueOf(this.enumType, source.trim());

}

}

}

3.5.3. GenericConverter

When you require a sophisticated Converter implementation, consider the GenericConverter interface. With a more flexible but less strongly typed signature, a GenericConverter supports converting between multiple source and target types. In addition, a GenericConverter makes available source and target field context you can use when implementing your conversion logic. Such context allows a type conversion to be driven by a field annotation, or generic information declared on a field signature.

package org.springframework.core.convert.converter;

public interface GenericConverter {

public Set<ConvertiblePair> getConvertibleTypes();

Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}

To implement a GenericConverter, have getConvertibleTypes() return the supported source→target type pairs. Then implement convert(Object, TypeDescriptor, TypeDescriptor) to implement your conversion logic. The source TypeDescriptor provides access to the source field holding the value being converted. The target TypeDescriptor provides access to the target field where the converted value will be set.

A good example of a GenericConverter is a converter that converts between a Java Array and a Collection. Such an ArrayToCollectionConverter introspects the field that declares the target Collection type to resolve the Collection’s element type. This allows each element in the source array to be converted to the Collection element type before the Collection is set on the target field.

Because GenericConverter is a more complex SPI interface, only use it when you need it. Favor Converter or ConverterFactory for basic type conversion needs.

ConditionalGenericConverter

Sometimes you only want a Converter to execute if a specific condition holds true. For example, you might only want to execute a Converter if a specific annotation is present on the target field. Or you might only want to execute a Converter if a specific method, such as a static valueOf method, is defined on the target class. ConditionalGenericConverter is the union of the GenericConverter and ConditionalConverter interfaces that allows you to define such custom matching criteria:

public interface ConditionalConverter {

boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType);

}

public interface ConditionalGenericConverter extends GenericConverter, ConditionalConverter {

}

A good example of a ConditionalGenericConverter is an EntityConverter that converts between an persistent entity identifier and an entity reference. Such a EntityConverter might only match if the target entity type declares a static finder method e.g. findAccount(Long). You would perform such a finder method check in the implementation of matches(TypeDescriptor, TypeDescriptor).

3.5.4. ConversionService API

The ConversionService defines a unified API for executing type conversion logic at runtime. Converters are often executed behind this facade interface:

package org.springframework.core.convert;

public interface ConversionService {

boolean canConvert(Class<?> sourceType, Class<?> targetType);

<T> T convert(Object source, Class<T> targetType);

boolean canConvert(TypeDescriptor sourceType, TypeDescriptor targetType);

Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor targetType);

}

Most ConversionService implementations also implement ConverterRegistry, which provides an SPI for registering converters. Internally, a ConversionService implementation delegates to its registered converters to carry out type conversion logic.

A robust ConversionService implementation is provided in the core.convert.support package. GenericConversionService is the general-purpose implementation suitable for use in most environments. ConversionServiceFactory provides a convenient factory for creating common ConversionService configurations.

3.5.5. Configuring a ConversionService

A ConversionService is a stateless object designed to be instantiated at application startup, then shared between multiple threads. In a Spring application, you typically configure a ConversionService instance per Spring container (or ApplicationContext). That ConversionService will be picked up by Spring and then used whenever a type conversion needs to be performed by the framework. You may also inject this ConversionService into any of your beans and invoke it directly.

If no ConversionService is registered with Spring, the original PropertyEditor-based system is used.

To register a default ConversionService with Spring, add the following bean definition with id conversionService:

<bean id="conversionService"

class="org.springframework.context.support.ConversionServiceFactoryBean"/>

A default ConversionService can convert between strings, numbers, enums, collections, maps, and other common types. To supplement or override the default converters with your own custom converter(s), set the converters property. Property values may implement either of the Converter, ConverterFactory, or GenericConverter interfaces.

<bean id="conversionService"

class="org.springframework.context.support.ConversionServiceFactoryBean">

<property name="converters">

<set>

<bean class="example.MyCustomConverter"/>

</set>

</property>

</bean>

It is also common to use a ConversionService within a Spring MVC application. See Conversion and Formatting in the Spring MVC chapter.

In certain situations you may wish to apply formatting during conversion. See FormatterRegistry SPI for details on using FormattingConversionServiceFactoryBean.

3.5.6. Using a ConversionService programmatically

To work with a ConversionService instance programmatically, simply inject a reference to it like you would for any other bean:

@Service

public class MyService {

@Autowired

public MyService(ConversionService conversionService) {

this.conversionService = conversionService;

}

public void doIt() {

this.conversionService.convert(...)

}

}

For most use cases, the convert method specifying the targetType can be used but it will not work with more complex types such as a collection of a parameterized element. If you want to convert a List of Integer to a List of String programmatically, for instance, you need to provide a formal definition of the source and target types.

Fortunately, TypeDescriptor provides various options to make that straightforward:

DefaultConversionService cs = new DefaultConversionService();

List<Integer> input = ....

cs.convert(input,

TypeDescriptor.forObject(input), // List<Integer> type descriptor

TypeDescriptor.collection(List.class, TypeDescriptor.valueOf(String.class)));

Note that DefaultConversionService registers converters automatically which are appropriate for most environments. This includes collection converters, scalar converters, and also basic Object to String converters. The same converters can be registered with any ConverterRegistry using the static addDefaultConverters method on the DefaultConversionService class.

Converters for value types will be reused for arrays and collections, so there is no need to create a specific converter to convert from a Collection of S to a Collection of T, assuming that standard collection handling is appropriate.

3.6. Spring Field Formatting

As discussed in the previous section, core.convert is a general-purpose type conversion system. It provides a unified ConversionService API as well as a strongly-typed Converter SPI for implementing conversion logic from one type to another. A Spring Container uses this system to bind bean property values. In addition, both the Spring Expression Language (SpEL) and DataBinder use this system to bind field values. For example, when SpEL needs to coerce a Short to a Long to complete an expression.setValue(Object bean, Object value) attempt, the core.convert system performs the coercion.

Now consider the type conversion requirements of a typical client environment such as a web or desktop application. In such environments, you typically convert from String to support the client postback process, as well as back to String to support the view rendering process. In addition, you often need to localize String values. The more general core.convert Converter SPI does not address such formatting requirements directly. To directly address them, Spring 3 introduces a convenient Formatter SPI that provides a simple and robust alternative to PropertyEditors for client environments.

In general, use the Converter SPI when you need to implement general-purpose type conversion logic; for example, for converting between a java.util.Date and and java.lang.Long. Use the Formatter SPI when you’re working in a client environment, such as a web application, and need to parse and print localized field values. The ConversionService provides a unified type conversion API for both SPIs.

3.6.1. Formatter SPI

The Formatter SPI to implement field formatting logic is simple and strongly typed:

package org.springframework.format;

public interface Formatter<T> extends Printer<T>, Parser<T> {

}

Where Formatter extends from the Printer and Parser building-block interfaces:

public interface Printer<T> {

String print(T fieldValue, Locale locale);

}

import java.text.ParseException;

public interface Parser<T> {

T parse(String clientValue, Locale locale) throws ParseException;

}

To create your own Formatter, simply implement the Formatter interface above. Parameterize T to be the type of object you wish to format, for example, java.util.Date. Implement the print() operation to print an instance of T for display in the client locale. Implement the parse() operation to parse an instance of T from the formatted representation returned from the client locale. Your Formatter should throw a ParseException or IllegalArgumentException if a parse attempt fails. Take care to ensure your Formatter implementation is thread-safe.

Several Formatter implementations are provided in format subpackages as a convenience. The number package provides a NumberStyleFormatter, CurrencyStyleFormatter, and PercentStyleFormatter to format java.lang.Number objects using a java.text.NumberFormat. The datetime package provides a DateFormatter to format java.util.Date objects with a java.text.DateFormat. The datetime.joda package provides comprehensive datetime formatting support based on the Joda-Time library.

Consider DateFormatter as an example Formatter implementation:

package org.springframework.format.datetime;

public final class DateFormatter implements Formatter<Date> {

private String pattern;

public DateFormatter(String pattern) {

this.pattern = pattern;

}

public String print(Date date, Locale locale) {

if (date == null) {

return "";

}

return getDateFormat(locale).format(date);

}

public Date parse(String formatted, Locale locale) throws ParseException {

if (formatted.length() == 0) {

return null;

}

return getDateFormat(locale).parse(formatted);

}

protected DateFormat getDateFormat(Locale locale) {

DateFormat dateFormat = new SimpleDateFormat(this.pattern, locale);

dateFormat.setLenient(false);

return dateFormat;

}

}

The Spring team welcomes community-driven Formatter contributions; see jira.spring.io to contribute.

3.6.2. Annotation-driven Formatting

As you will see, field formatting can be configured by field type or annotation. To bind an Annotation to a formatter, implement AnnotationFormatterFactory:

package org.springframework.format;

public interface AnnotationFormatterFactory<A extends Annotation> {

Set<Class<?>> getFieldTypes();

Printer<?> getPrinter(A annotation, Class<?> fieldType);

Parser<?> getParser(A annotation, Class<?> fieldType);

}

Parameterize A to be the field annotationType you wish to associate formatting logic with, for example org.springframework.format.annotation.DateTimeFormat. Have getFieldTypes() return the types of fields the annotation may be used on. Have getPrinter() return a Printer to print the value of an annotated field. Have getParser() return a Parser to parse a clientValue for an annotated field.

The example AnnotationFormatterFactory implementation below binds the @NumberFormat Annotation to a formatter. This annotation allows either a number style or pattern to be specified:

public final class NumberFormatAnnotationFormatterFactory

implements AnnotationFormatterFactory<NumberFormat> {

public Set<Class<?>> getFieldTypes() {

return new HashSet<Class<?>>(asList(new Class<?>[] {

Short.class, Integer.class, Long.class, Float.class,

Double.class, BigDecimal.class, BigInteger.class }));

}

public Printer<Number> getPrinter(NumberFormat annotation, Class<?> fieldType) {

return configureFormatterFrom(annotation, fieldType);

}

public Parser<Number> getParser(NumberFormat annotation, Class<?> fieldType) {

return configureFormatterFrom(annotation, fieldType);

}

private Formatter<Number> configureFormatterFrom(NumberFormat annotation, Class<?> fieldType) {

if (!annotation.pattern().isEmpty()) {

return new NumberStyleFormatter(annotation.pattern());

} else {

Style style = annotation.style();

if (style == Style.PERCENT) {

return new PercentStyleFormatter();

} else if (style == Style.CURRENCY) {

return new CurrencyStyleFormatter();

} else {

return new NumberStyleFormatter();

}

}

}

}

To trigger formatting, simply annotate fields with @NumberFormat:

public class MyModel {

@NumberFormat(style=Style.CURRENCY)

private BigDecimal decimal;

}

Format Annotation API

A portable format annotation API exists in the org.springframework.format.annotation package. Use @NumberFormat to format java.lang.Number fields. Use @DateTimeFormat to format java.util.Date, java.util.Calendar, java.util.Long, or Joda-Time fields.

The example below uses @DateTimeFormat to format a java.util.Date as a ISO Date (yyyy-MM-dd):

public class MyModel {

@DateTimeFormat(iso=ISO.DATE)

private Date date;

}

3.6.3. FormatterRegistry SPI

The FormatterRegistry is an SPI for registering formatters and converters. FormattingConversionService is an implementation of FormatterRegistry suitable for most environments. This implementation may be configured programmatically or declaratively as a Spring bean using FormattingConversionServiceFactoryBean. Because this implementation also implements ConversionService, it can be directly configured for use with Spring’s DataBinder and the Spring Expression Language (SpEL).

Review the FormatterRegistry SPI below:

package org.springframework.format;

public interface FormatterRegistry extends ConverterRegistry {

void addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> parser);

void addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter);

void addFormatterForFieldType(Formatter<?> formatter);

void addFormatterForAnnotation(AnnotationFormatterFactory<?, ?> factory);

}

As shown above, Formatters can be registered by fieldType or annotation.

The FormatterRegistry SPI allows you to configure Formatting rules centrally, instead of duplicating such configuration across your Controllers. For example, you might want to enforce that all Date fields are formatted a certain way, or fields with a specific annotation are formatted in a certain way. With a shared FormatterRegistry, you define these rules once and they are applied whenever formatting is needed.

3.6.4. FormatterRegistrar SPI

The FormatterRegistrar is an SPI for registering formatters and converters through the FormatterRegistry:

package org.springframework.format;

public interface FormatterRegistrar {

void registerFormatters(FormatterRegistry registry);

}

A FormatterRegistrar is useful when registering multiple related converters and formatters for a given formatting category, such as Date formatting. It can also be useful where declarative registration is insufficient. For example when a formatter needs to be indexed under a specific field type different from its own <T> or when registering a Printer/Parser pair. The next section provides more information on converter and formatter registration.

3.6.5. Configuring Formatting in Spring MVC

See Conversion and Formatting in the Spring MVC chapter.

3.7. Configuring a global date & time format

By default, date and time fields that are not annotated with @DateTimeFormat are converted from strings using the DateFormat.SHORT style. If you prefer, you can change this by defining your own global format.

You will need to ensure that Spring does not register default formatters, and instead you should register all formatters manually. Use the org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar or org.springframework.format.datetime.DateFormatterRegistrar class depending on whether you use the Joda-Time library.

For example, the following Java configuration will register a global ' `yyyyMMdd’ format. This example does not depend on the Joda-Time library:

@Configuration

public class AppConfig {

@Bean

public FormattingConversionService conversionService() {

// Use the DefaultFormattingConversionService but do not register defaults

DefaultFormattingConversionService conversionService = new DefaultFormattingConversionService(false);

// Ensure @NumberFormat is still supported

conversionService.addFormatterForFieldAnnotation(new NumberFormatAnnotationFormatterFactory());

// Register date conversion with a specific global format

DateFormatterRegistrar registrar = new DateFormatterRegistrar();

registrar.setFormatter(new DateFormatter("yyyyMMdd"));

registrar.registerFormatters(conversionService);

return conversionService;

}

}

If you prefer XML based configuration you can use a FormattingConversionServiceFactoryBean. Here is the same example, this time using Joda Time:

<?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="conversionService" class="org.springframework.format.support.FormattingConversionServiceFactoryBean">

<property name="registerDefaultFormatters" value="false" />

<property name="formatters">

<set>

<bean class="org.springframework.format.number.NumberFormatAnnotationFormatterFactory" />

</set>

</property>

<property name="formatterRegistrars">

<set>

<bean class="org.springframework.format.datetime.joda.JodaTimeFormatterRegistrar">

<property name="dateFormatter">

<bean class="org.springframework.format.datetime.joda.DateTimeFormatterFactoryBean">

<property name="pattern" value="yyyyMMdd"/>

</bean>

</property>

</bean>

</set>

</property>

</bean>

</beans>

Joda-Time provides separate distinct types to represent date, time and date-time values. The dateFormatter, timeFormatter and dateTimeFormatter properties of the JodaTimeFormatterRegistrar should be used to configure the different formats for each type. The DateTimeFormatterFactoryBean provides a convenient way to create formatters.

If you are using Spring MVC remember to explicitly configure the conversion service that is used. For Java based @Configuration this means extending the WebMvcConfigurationSupport class and overriding the mvcConversionService() method. For XML you should use the 'conversion-service' attribute of the mvc:annotation-driven element. See Conversion and Formatting for details.

3.8. Spring Validation

Spring 3 introduces several enhancements to its validation support. First, the JSR-303 Bean Validation API is now fully supported. Second, when used programmatically, Spring’s DataBinder can now validate objects as well as bind to them. Third, Spring MVC now has support for declaratively validating @Controller inputs.

3.8.1. Overview of the JSR-303 Bean Validation API

JSR-303 standardizes validation constraint declaration and metadata for the Java platform. Using this API, you annotate domain model properties with declarative validation constraints and the runtime enforces them. There are a number of built-in constraints you can take advantage of. You may also define your own custom constraints.

To illustrate, consider a simple PersonForm model with two properties:

public class PersonForm {

private String name;

private int age;

}

JSR-303 allows you to define declarative validation constraints against such properties:

public class PersonForm {

@NotNull

@Size(max=64)

private String name;

@Min(0)

private int age;

}

When an instance of this class is validated by a JSR-303 Validator, these constraints will be enforced.

For general information on JSR-303/JSR-349, see the Bean Validation website. For information on the specific capabilities of the default reference implementation, see the Hibernate Validator documentation. To learn how to setup a Bean Validation provider as a Spring bean, keep reading.

3.8.2. Configuring a Bean Validation Provider

Spring provides full support for the Bean Validation API. This includes convenient support for bootstrapping a JSR-303/JSR-349 Bean Validation provider as a Spring bean. This allows for a javax.validation.ValidatorFactory or javax.validation.Validator to be injected wherever validation is needed in your application.

Use the LocalValidatorFactoryBean to configure a default Validator as a Spring bean:

<bean id="validator"

class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"/>

The basic configuration above will trigger Bean Validation to initialize using its default bootstrap mechanism. A JSR-303/JSR-349 provider, such as Hibernate Validator, is expected to be present in the classpath and will be detected automatically.

Injecting a Validator

LocalValidatorFactoryBean implements both javax.validation.ValidatorFactory and javax.validation.Validator, as well as Spring’s org.springframework.validation.Validator. You may inject a reference to either of these interfaces into beans that need to invoke validation logic.

Inject a reference to javax.validation.Validator if you prefer to work with the Bean Validation API directly:

import javax.validation.Validator;

@Service

public class MyService {

@Autowired

private Validator validator;

Inject a reference to org.springframework.validation.Validator if your bean requires the Spring Validation API:

import org.springframework.validation.Validator;

@Service

public class MyService {

@Autowired

private Validator validator;

}

Configuring Custom Constraints

Each Bean Validation constraint consists of two parts. First, a @Constraint annotation that declares the constraint and its configurable properties. Second, an implementation of the javax.validation.ConstraintValidator interface that implements the constraint’s behavior. To associate a declaration with an implementation, each @Constraint annotation references a corresponding ConstraintValidator implementation class. At runtime, a ConstraintValidatorFactory instantiates the referenced implementation when the constraint annotation is encountered in your domain model.

By default, the LocalValidatorFactoryBean configures a SpringConstraintValidatorFactory that uses Spring to create ConstraintValidator instances. This allows your custom ConstraintValidators to benefit from dependency injection like any other Spring bean.

Shown below is an example of a custom @Constraint declaration, followed by an associated ConstraintValidator implementation that uses Spring for dependency injection:

@Target({ElementType.METHOD, ElementType.FIELD})

@Retention(RetentionPolicy.RUNTIME)

@Constraint(validatedBy=MyConstraintValidator.class)

public @interface MyConstraint {

}

import javax.validation.ConstraintValidator;

public class MyConstraintValidator implements ConstraintValidator {

@Autowired;

private Foo aDependency;

...

}

As you can see, a ConstraintValidator implementation may have its dependencies @Autowired like any other Spring bean.

Spring-driven Method Validation

The method validation feature supported by Bean Validation 1.1, and as a custom extension also by Hibernate Validator 4.3, can be integrated into a Spring context through a MethodValidationPostProcessor bean definition:

<bean class="org.springframework.validation.beanvalidation.MethodValidationPostProcessor"/>

In order to be eligible for Spring-driven method validation, all target classes need to be annotated with Spring’s @Validated annotation, optionally declaring the validation groups to use. Check out the MethodValidationPostProcessor javadocs for setup details with Hibernate Validator and Bean Validation 1.1 providers.

Additional Configuration Options

The default LocalValidatorFactoryBean configuration should prove sufficient for most cases. There are a number of configuration options for various Bean Validation constructs, from message interpolation to traversal resolution. See the LocalValidatorFactoryBean javadocs for more information on these options.

3.8.3. Configuring a DataBinder

Since Spring 3, a DataBinder instance can be configured with a Validator. Once configured, the Validator may be invoked by calling binder.validate(). Any validation Errors are automatically added to the binder’s BindingResult.

When working with the DataBinder programmatically, this can be used to invoke validation logic after binding to a target object:

Foo target = new Foo();

DataBinder binder = new DataBinder(target);

binder.setValidator(new FooValidator());

// bind to the target object

binder.bind(propertyValues);

// validate the target object

binder.validate();

// get BindingResult that includes any validation errors

BindingResult results = binder.getBindingResult();

A DataBinder can also be configured with multiple Validator instances via dataBinder.addValidators and dataBinder.replaceValidators. This is useful when combining globally configured Bean Validation with a Spring Validator configured locally on a DataBinder instance. See [validation-mvc-configuring].

3.8.4. Spring MVC 3 Validation

See Validation in the Spring MVC chapter.

4. Spring Expression Language (SpEL)

4.1. Introduction

The Spring Expression Language (SpEL for short) is a powerful expression language that supports querying and manipulating an object graph at runtime. The language syntax is similar to Unified EL but offers additional features, most notably method invocation and basic string templating functionality.

While there are several other Java expression languages available — OGNL, MVEL, and JBoss EL, to name a few — the Spring Expression Language was created to provide the Spring community with a single well supported expression language that can be used across all the products in the Spring portfolio. Its language features are driven by the requirements of the projects in the Spring portfolio, including tooling requirements for code completion support within the Eclipse based Spring Tool Suite. That said, SpEL is based on a technology agnostic API allowing other expression language implementations to be integrated should the need arise.

While SpEL serves as the foundation for expression evaluation within the Spring portfolio, it is not directly tied to Spring and can be used independently. In order to be self contained, many of the examples in this chapter use SpEL as if it were an independent expression language. This requires creating a few bootstrapping infrastructure classes such as the parser. Most Spring users will not need to deal with this infrastructure and will instead only author expression strings for evaluation. An example of this typical use is the integration of SpEL into creating XML or annotated based bean definitions as shown in the section Expression support for defining bean definitions.

This chapter covers the features of the expression language, its API, and its language syntax. In several places an Inventor and Inventor’s Society classes are used as the target objects for expression evaluation. These class declarations and the data used to populate them are listed at the end of the chapter.

The expression language supports the following functionality:

Literal expressions

Boolean and relational operators

Regular expressions

Class expressions

Accessing properties, arrays, lists, maps

Method invocation

Relational operators

Assignment

Calling constructors

Bean references

Array construction

Inline lists

Inline maps

Ternary operator

Variables

User defined functions

Collection projection

Collection selection

Templated expressions

4.2. Evaluation

This section introduces the simple use of SpEL interfaces and its expression language. The complete language reference can be found in the section Language Reference.

The following code introduces the SpEL API to evaluate the literal string expression 'Hello World'.

ExpressionParser parser = new SpelExpressionParser();

Expression exp = parser.parseExpression("'Hello World'");

String message = (String) exp.getValue();

The value of the message variable is simply 'Hello World'.

The SpEL classes and interfaces you are most likely to use are located in the packages org.springframework.expression and its sub packages such as spel.support.

The interface ExpressionParser is responsible for parsing an expression string. In this example the expression string is a string literal denoted by the surrounding single quotes. The interface Expression is responsible for evaluating the previously defined expression string. There are two exceptions that can be thrown, ParseException and EvaluationException when calling parser.parseExpression and exp.getValue respectively.

SpEL supports a wide range of features, such as calling methods, accessing properties, and calling constructors.

As an example of method invocation, we call the concat method on the string literal.

ExpressionParser parser = new SpelExpressionParser();

Expression exp = parser.parseExpression("'Hello World'.concat('!')");

String message = (String) exp.getValue();

The value of message is now 'Hello World!'.

As an example of calling a JavaBean property, the String property Bytes can be called as shown below.

ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes()'

Expression exp = parser.parseExpression("'Hello World'.bytes");

byte[] bytes = (byte[]) exp.getValue();

SpEL also supports nested properties using standard dot notation, i.e. prop1.prop2.prop3 and the setting of property values

Public fields may also be accessed.

ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes().length'

Expression exp = parser.parseExpression("'Hello World'.bytes.length");

int length = (Integer) exp.getValue();

The String’s constructor can be called instead of using a string literal.

ExpressionParser parser = new SpelExpressionParser();

Expression exp = parser.parseExpression("new String('hello world').toUpperCase()");

String message = exp.getValue(String.class);

Note the use of the generic method public <T> T getValue(Class<T> desiredResultType). Using this method removes the need to cast the value of the expression to the desired result type. An EvaluationException will be thrown if the value cannot be cast to the type T or converted using the registered type converter.

The more common usage of SpEL is to provide an expression string that is evaluated against a specific object instance (called the root object). The example shows how to retrieve the name property from an instance of the Inventor class or create a boolean condition:

// Create and set a calendar

GregorianCalendar c = new GregorianCalendar();

c.set(1856, 7, 9);

// The constructor arguments are name, birthday, and nationality.

Inventor tesla = new Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = new SpelExpressionParser();

Expression exp = parser.parseExpression("name");

String name = (String) exp.getValue(tesla);

// name == "Nikola Tesla"

exp = parser.parseExpression("name == 'Nikola Tesla'");

boolean result = exp.getValue(tesla, Boolean.class);

// result == true

4.2.1. EvaluationContext

The interface EvaluationContext is used when evaluating an expression to resolve properties, methods, or fields and to help perform type conversion. There are two out-of-the-box implementations.

SimpleEvaluationContext — exposes a subset of essential SpEL language features and configuration options, for categories of expressions that do not require the full extent of the SpEL language syntax and should be meaningfully restricted. Examples include but are not limited to data binding expressions, property-based filters, and others.

StandardEvaluationContext — exposes the full set of SpEL language features and configuration options. You may use it to specify a default root object and to configure every available evaluation-related strategy.

SimpleEvaluationContext is designed to support only a subset of the SpEL language syntax. It excludes Java type references, constructors, and bean references. It also requires that one explicitly choose the level of support for properties and methods in expressions. By default, the create() static factory method enables only read access to properties. You can also obtain a builder to configure the exact level of support needed, targeting one or some combination of the following:

Custom PropertyAccessor only (no reflection)

Data binding properties for read-only access

Data binding properties for read and write

Type conversion

By default SpEL uses the conversion service available in Spring core (org.springframework.core.convert.ConversionService). This conversion service comes with many converters built in for common conversions but is also fully extensible so custom conversions between types can be added. Additionally it has the key capability that it is generics aware. This means that when working with generic types in expressions, SpEL will attempt conversions to maintain type correctness for any objects it encounters.

What does this mean in practice? Suppose assignment, using setValue(), is being used to set a List property. The type of the property is actually List<Boolean>. SpEL will recognize that the elements of the list need to be converted to Boolean before being placed in it. A simple example:

class Simple {

public List<Boolean> booleanList = new ArrayList<Boolean>();

}

Simple simple = new Simple();

simple.booleanList.add(true);

EvaluationContext context = SimpleEvaluationContext().forReadOnlyDataBinding().build();

// false is passed in here as a string. SpEL and the conversion service will

// correctly recognize that it needs to be a Boolean and convert it

parser.parseExpression("booleanList[0]").setValue(context, simple, "false");

// b will be false

Boolean b = simple.booleanList.get(0);

4.2.2. Parser configuration

It is possible to configure the SpEL expression parser using a parser configuration object (org.springframework.expression.spel.SpelParserConfiguration). The configuration object controls the behavior of some of the expression components. For example, if indexing into an array or collection and the element at the specified index is null it is possible to automatically create the element. This is useful when using expressions made up of a chain of property references. If indexing into an array or list and specifying an index that is beyond the end of the current size of the array or list it is possible to automatically grow the array or list to accommodate that index.

class Demo {

public List<String> list;

}

// Turn on:

// - auto null reference initialization

// - auto collection growing

SpelParserConfiguration config = new SpelParserConfiguration(true,true);

ExpressionParser parser = new SpelExpressionParser(config);

Expression expression = parser.parseExpression("list[3]");

Demo demo = new Demo();

Object o = expression.getValue(demo);

// demo.list will now be a real collection of 4 entries

// Each entry is a new empty String

It is also possible to configure the behaviour of the SpEL expression compiler.

4.2.3. SpEL compilation

Spring Framework 4.1 includes a basic expression compiler. Expressions are usually interpreted which provides a lot of dynamic flexibility during evaluation but does not provide optimum performance. For occasional expression usage this is fine, but when used by other components like Spring Integration, performance can be very important and there is no real need for the dynamism.

The SpEL compiler is intended to address this need. The compiler will generate a real Java class on the fly during evaluation that embodies the expression behavior and use that to achieve much faster expression evaluation. Due to the lack of typing around expressions the compiler uses information gathered during the interpreted evaluations of an expression when performing compilation. For example, it does not know the type of a property reference purely from the expression, but during the first interpreted evaluation it will find out what it is. Of course, basing the compilation on this information could cause trouble later if the types of the various expression elements change over time. For this reason compilation is best suited to expressions whose type information is not going to change on repeated evaluations.

For a basic expression like this:

someArray[0].someProperty.someOtherProperty < 0.1

which involves array access, some property derefencing and numeric operations, the performance gain can be very noticeable. In an example micro benchmark run of 50000 iterations, it was taking 75ms to evaluate using only the interpreter and just 3ms using the compiled version of the expression.

Compiler configuration

The compiler is not turned on by default, but there are two ways to turn it on. It can be turned on using the parser configuration process discussed earlier or via a system property when SpEL usage is embedded inside another component. This section discusses both of these options.

It is important to understand that there are a few modes the compiler can operate in, captured in an enum (org.springframework.expression.spel.SpelCompilerMode). The modes are as follows:

OFF - The compiler is switched off; this is the default.

IMMEDIATE - In immediate mode the expressions are compiled as soon as possible. This is typically after the first interpreted evaluation. If the compiled expression fails (typically due to a type changing, as described above) then the caller of the expression evaluation will receive an exception.

MIXED - In mixed mode the expressions silently switch between interpreted and compiled mode over time. After some number of interpreted runs they will switch to compiled form and if something goes wrong with the compiled form (like a type changing, as described above) then the expression will automatically switch back to interpreted form again. Sometime later it may generate another compiled form and switch to it. Basically the exception that the user gets in IMMEDIATE mode is instead handled internally.

IMMEDIATE mode exists because MIXED mode could cause issues for expressions that have side effects. If a compiled expression blows up after partially succeeding it may have already done something that has affected the state of the system. If this has happened the caller may not want it to silently re-run in interpreted mode since part of the expression may be running twice.

After selecting a mode, use the SpelParserConfiguration to configure the parser:

SpelParserConfiguration config = new SpelParserConfiguration(SpelCompilerMode.IMMEDIATE,

this.getClass().getClassLoader());

SpelExpressionParser parser = new SpelExpressionParser(config);

Expression expr = parser.parseExpression("payload");

MyMessage message = new MyMessage();

Object payload = expr.getValue(message);

When specifying the compiler mode it is also possible to specify a classloader (passing null is allowed). Compiled expressions will be defined in a child classloader created under any that is supplied. It is important to ensure if a classloader is specified it can see all the types involved in the expression evaluation process. If none is specified then a default classloader will be used (typically the context classloader for the thread that is running during expression evaluation).

The second way to configure the compiler is for use when SpEL is embedded inside some other component and it may not be possible to configure via a configuration object. In these cases it is possible to use a system property. The property spring.expression.compiler.mode can be set to one of the SpelCompilerMode enum values (off, immediate, or mixed).

Compiler limitations

Since Spring Framework 4.1 the basic compilation framework is in place. However, the framework does not yet support compiling every kind of expression. The initial focus has been on the common expressions that are likely to be used in performance critical contexts. The following kinds of expression cannot be compiled at the moment:

expressions involving assignment

expressions relying on the conversion service

expressions using custom resolvers or accessors

expressions using selection or projection

More and more types of expression will be compilable in the future.

4.3. Expressions in bean definitions

SpEL expressions can be used with XML or annotation-based configuration metadata for defining BeanDefinitions. In both cases the syntax to define the expression is of the form #{ <expression string> }.

4.3.1. XML configuration

A property or constructor-arg value can be set using expressions as shown below.

<bean id="numberGuess" class="org.spring.samples.NumberGuess">

<property name="randomNumber" value="#{ T(java.lang.Math).random() \* 100.0 }"/>

<!-- other properties -->

</bean>

The variable systemProperties is predefined, so you can use it in your expressions as shown below. Note that you do not have to prefix the predefined variable with the # symbol in this context.

<bean id="taxCalculator" class="org.spring.samples.TaxCalculator">

<property name="defaultLocale" value="#{ systemProperties['user.region'] }"/>

<!-- other properties -->

</bean>

You can also refer to other bean properties by name, for example.

<bean id="numberGuess" class="org.spring.samples.NumberGuess">

<property name="randomNumber" value="#{ T(java.lang.Math).random() \* 100.0 }"/>

<!-- other properties -->

</bean>

<bean id="shapeGuess" class="org.spring.samples.ShapeGuess">

<property name="initialShapeSeed" value="#{ numberGuess.randomNumber }"/>

<!-- other properties -->

</bean>

4.3.2. Annotation config

The @Value annotation can be placed on fields, methods and method/constructor parameters to specify a default value.

Here is an example to set the default value of a field variable.

public static class FieldValueTestBean

@Value("#{ systemProperties['user.region'] }")

private String defaultLocale;

public void setDefaultLocale(String defaultLocale) {

this.defaultLocale = defaultLocale;

}

public String getDefaultLocale() {

return this.defaultLocale;

}

}

The equivalent but on a property setter method is shown below.

public static class PropertyValueTestBean

private String defaultLocale;

@Value("#{ systemProperties['user.region'] }")

public void setDefaultLocale(String defaultLocale) {

this.defaultLocale = defaultLocale;

}

public String getDefaultLocale() {

return this.defaultLocale;

}

}

Autowired methods and constructors can also use the @Value annotation.

public class SimpleMovieLister {

private MovieFinder movieFinder;

private String defaultLocale;

@Autowired

public void configure(MovieFinder movieFinder,

@Value("#{ systemProperties['user.region'] }") String defaultLocale) {

this.movieFinder = movieFinder;

this.defaultLocale = defaultLocale;

}

// ...

}

public class MovieRecommender {

private String defaultLocale;

private CustomerPreferenceDao customerPreferenceDao;

@Autowired

public MovieRecommender(CustomerPreferenceDao customerPreferenceDao,

@Value("#{systemProperties['user.country']}") String defaultLocale) {

this.customerPreferenceDao = customerPreferenceDao;

this.defaultLocale = defaultLocale;

}

// ...

}

4.4. Language Reference

4.4.1. Literal expressions

The types of literal expressions supported are strings, numeric values (int, real, hex), boolean and null. Strings are delimited by single quotes. To put a single quote itself in a string, use two single quote characters.

The following listing shows simple usage of literals. Typically they would not be used in isolation like this but rather as part of a more complex expression, for example using a literal on one side of a logical comparison operator.

ExpressionParser parser = new SpelExpressionParser();

// evals to "Hello World"

String helloWorld = (String) parser.parseExpression("'Hello World'").getValue();

double avogadrosNumber = (Double) parser.parseExpression("6.0221415E+23").getValue();

// evals to 2147483647

int maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").getValue();

boolean trueValue = (Boolean) parser.parseExpression("true").getValue();

Object nullValue = parser.parseExpression("null").getValue();

Numbers support the use of the negative sign, exponential notation, and decimal points. By default real numbers are parsed using Double.parseDouble().

4.4.2. Properties, Arrays, Lists, Maps, Indexers

Navigating with property references is easy: just use a period to indicate a nested property value. The instances of the Inventor class, pupin, and tesla, were populated with data listed in the section Classes used in the examples. To navigate "down" and get Tesla’s year of birth and Pupin’s city of birth the following expressions are used.

// evals to 1856

int year = (Integer) parser.parseExpression("Birthdate.Year + 1900").getValue(context);

String city = (String) parser.parseExpression("placeOfBirth.City").getValue(context);

Case insensitivity is allowed for the first letter of property names. The contents of arrays and lists are obtained using square bracket notation.

ExpressionParser parser = new SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

// Inventions Array

// evaluates to "Induction motor"

String invention = parser.parseExpression("inventions[3]").getValue(

context, tesla, String.class);

// Members List

// evaluates to "Nikola Tesla"

String name = parser.parseExpression("Members[0].Name").getValue(

context, ieee, String.class);

// List and Array navigation

// evaluates to "Wireless communication"

String invention = parser.parseExpression("Members[0].Inventions[6]").getValue(

context, ieee, String.class);

The contents of maps are obtained by specifying the literal key value within the brackets. In this case, because keys for the Officers map are strings, we can specify string literals.

// Officer's Dictionary

Inventor pupin = parser.parseExpression("Officers['president']").getValue(

societyContext, Inventor.class);

// evaluates to "Idvor"

String city = parser.parseExpression("Officers['president'].PlaceOfBirth.City").getValue(

societyContext, String.class);

// setting values

parser.parseExpression("Officers['advisors'][0].PlaceOfBirth.Country").setValue(

societyContext, "Croatia");

4.4.3. Inline lists

Lists can be expressed directly in an expression using {} notation.

// evaluates to a Java list containing the four numbers

List numbers = (List) parser.parseExpression("{1,2,3,4}").getValue(context);

List listOfLists = (List) parser.parseExpression("{{'a','b'},{'x','y'}}").getValue(context);

{} by itself means an empty list. For performance reasons, if the list is itself entirely composed of fixed literals then a constant list is created to represent the expression, rather than building a new list on each evaluation.

4.4.4. Inline Maps

Maps can also be expressed directly in an expression using {key:value} notation.

// evaluates to a Java map containing the two entries

Map inventorInfo = (Map) parser.parseExpression("{name:'Nikola',dob:'10-July-1856'}").getValue(context);

Map mapOfMaps = (Map) parser.parseExpression("{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',year:1856}}").getValue(context);

{:} by itself means an empty map. For performance reasons, if the map is itself composed of fixed literals or other nested constant structures (lists or maps) then a constant map is created to represent the expression, rather than building a new map on each evaluation. Quoting of the map keys is optional, the examples above are not using quoted keys.

4.4.5. Array construction

Arrays can be built using the familiar Java syntax, optionally supplying an initializer to have the array populated at construction time.

int[] numbers1 = (int[]) parser.parseExpression("new int[4]").getValue(context);

// Array with initializer

int[] numbers2 = (int[]) parser.parseExpression("new int[]{1,2,3}").getValue(context);

// Multi dimensional array

int[][] numbers3 = (int[][]) parser.parseExpression("new int[4][5]").getValue(context);

It is not currently allowed to supply an initializer when constructing a multi-dimensional array.

4.4.6. Methods

Methods are invoked using typical Java programming syntax. You may also invoke methods on literals. Varargs are also supported.

// string literal, evaluates to "bc"

String bc = parser.parseExpression("'abc'.substring(1, 3)").getValue(String.class);

// evaluates to true

boolean isMember = parser.parseExpression("isMember('Mihajlo Pupin')").getValue(

societyContext, Boolean.class);

4.4.7. Operators

Relational operators

The relational operators; equal, not equal, less than, less than or equal, greater than, and greater than or equal are supported using standard operator notation.

// evaluates to true

boolean trueValue = parser.parseExpression("2 == 2").getValue(Boolean.class);

// evaluates to false

boolean falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean.class);

// evaluates to true

boolean trueValue = parser.parseExpression("'black' < 'block'").getValue(Boolean.class);

Greater/less-than comparisons against null follow a simple rule: null is treated as nothing here (i.e. NOT as zero). As a consequence, any other value is always greater than null (X > null is always true) and no other value is ever less than nothing (X < null is always false).

If you prefer numeric comparisons instead, please avoid number-based null comparisons in favor of comparisons against zero (e.g. X > 0 or X < 0).

In addition to standard relational operators SpEL supports the instanceof and regular expression based matches operator.

// evaluates to false

boolean falseValue = parser.parseExpression(

"'xyz' instanceof T(Integer)").getValue(Boolean.class);

// evaluates to true

boolean trueValue = parser.parseExpression(

"'5.00' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);

//evaluates to false

boolean falseValue = parser.parseExpression(

"'5.0067' matches '^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class);

Be careful with primitive types as they are immediately boxed up to the wrapper type, so 1 instanceof T(int) evaluates to false while 1 instanceof T(Integer) evaluates to true, as expected.

Each symbolic operator can also be specified as a purely alphabetic equivalent. This avoids problems where the symbols used have special meaning for the document type in which the expression is embedded (eg. an XML document). The textual equivalents are shown here: lt (<), gt (>), le (<=), ge (>=), eq (==), ne (!=), div (/), mod (%), not (!). These are case insensitive.

Logical operators

The logical operators that are supported are and, or, and not. Their use is demonstrated below.

// -- AND --

// evaluates to false

boolean falseValue = parser.parseExpression("true and false").getValue(Boolean.class);

// evaluates to true

String expression = "isMember('Nikola Tesla') and isMember('Mihajlo Pupin')";

boolean trueValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

// -- OR --

// evaluates to true

boolean trueValue = parser.parseExpression("true or false").getValue(Boolean.class);

// evaluates to true

String expression = "isMember('Nikola Tesla') or isMember('Albert Einstein')";

boolean trueValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

// -- NOT --

// evaluates to false

boolean falseValue = parser.parseExpression("!true").getValue(Boolean.class);

// -- AND and NOT --

String expression = "isMember('Nikola Tesla') and !isMember('Mihajlo Pupin')";

boolean falseValue = parser.parseExpression(expression).getValue(societyContext, Boolean.class);

Mathematical operators

The addition operator can be used on both numbers and strings. Subtraction, multiplication and division can be used only on numbers. Other mathematical operators supported are modulus (%) and exponential power (^). Standard operator precedence is enforced. These operators are demonstrated below.

// Addition

int two = parser.parseExpression("1 + 1").getValue(Integer.class); // 2

String testString = parser.parseExpression(

"'test' + ' ' + 'string'").getValue(String.class); // 'test string'

// Subtraction

int four = parser.parseExpression("1 - -3").getValue(Integer.class); // 4

double d = parser.parseExpression("1000.00 - 1e4").getValue(Double.class); // -9000

// Multiplication

int six = parser.parseExpression("-2 \* -3").getValue(Integer.class); // 6

double twentyFour = parser.parseExpression("2.0 \* 3e0 \* 4").getValue(Double.class); // 24.0

// Division

int minusTwo = parser.parseExpression("6 / -3").getValue(Integer.class); // -2

double one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double.class); // 1.0

// Modulus

int three = parser.parseExpression("7 % 4").getValue(Integer.class); // 3

int one = parser.parseExpression("8 / 5 % 2").getValue(Integer.class); // 1

// Operator precedence

int minusTwentyOne = parser.parseExpression("1+2-3\*8").getValue(Integer.class); // -21

4.4.8. Assignment

Setting of a property is done by using the assignment operator. This would typically be done within a call to setValue but can also be done inside a call to getValue.

Inventor inventor = new Inventor();

EvaluationContext context = SimpleEvaluationContext.forReadWriteDataBinding().build();

parser.parseExpression("Name").setValue(context, inventor, "Aleksandar Seovic");

// alternatively

String aleks = parser.parseExpression(

"Name = 'Aleksandar Seovic'").getValue(context, inventor, String.class);

4.4.9. Types

The special T operator can be used to specify an instance of java.lang.Class (the type). Static methods are invoked using this operator as well. The StandardEvaluationContext uses a TypeLocator to find types and the StandardTypeLocator (which can be replaced) is built with an understanding of the java.lang package. This means T() references to types within java.lang do not need to be fully qualified, but all other type references must be.

Class dateClass = parser.parseExpression("T(java.util.Date)").getValue(Class.class);

Class stringClass = parser.parseExpression("T(String)").getValue(Class.class);

boolean trueValue = parser.parseExpression(

"T(java.math.RoundingMode).CEILING < T(java.math.RoundingMode).FLOOR")

.getValue(Boolean.class);

4.4.10. Constructors

Constructors can be invoked using the new operator. The fully qualified class name should be used for all but the primitive type and String (where int, float, etc, can be used).

Inventor einstein = p.parseExpression(

"new org.spring.samples.spel.inventor.Inventor('Albert Einstein', 'German')")

.getValue(Inventor.class);

//create new inventor instance within add method of List

p.parseExpression(

"Members.add(new org.spring.samples.spel.inventor.Inventor(

'Albert Einstein', 'German'))").getValue(societyContext);

4.4.11. Variables

Variables can be referenced in the expression using the syntax #variableName. Variables are set using the method setVariable on EvaluationContext implementations.

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");

EvaluationContext context = SimpleEvaluationContext.forReadWriteDataBinding().build();

context.setVariable("newName", "Mike Tesla");

parser.parseExpression("Name = #newName").getValue(context, tesla);

System.out.println(tesla.getName()) // "Mike Tesla"

The #this and #root variables

The variable #this is always defined and refers to the current evaluation object (against which unqualified references are resolved). The variable #root is always defined and refers to the root context object. Although #this may vary as components of an expression are evaluated, #root always refers to the root.

// create an array of integers

List<Integer> primes = new ArrayList<Integer>();

primes.addAll(Arrays.asList(2,3,5,7,11,13,17));

// create parser and set variable 'primes' as the array of integers

ExpressionParser parser = new SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataAccess();

context.setVariable("primes", primes);

// all prime numbers > 10 from the list (using selection ?{...})

// evaluates to [11, 13, 17]

List<Integer> primesGreaterThanTen = (List<Integer>) parser.parseExpression(

"#primes.?[#this>10]").getValue(context);

4.4.12. Functions

You can extend SpEL by registering user defined functions that can be called within the expression string. The function is registered through the EvaluationContext.

Method method = ...;

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

context.setVariable("myFunction", method);

For example, given a utility method to reverse a string is shown below:

public abstract class StringUtils {

public static String reverseString(String input) {

StringBuilder backwards = new StringBuilder(input.length());

for (int i = 0; i < input.length(); i++)

backwards.append(input.charAt(input.length() - 1 - i));

}

return backwards.toString();

}

}

The above method can then be registered and used as follows:

ExpressionParser parser = new SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

context.setVariable("reverseString",

StringUtils.class.getDeclaredMethod("reverseString", String.class));

String helloWorldReversed = parser.parseExpression(

"#reverseString('hello')").getValue(context, String.class);

4.4.13. Bean references

If the evaluation context has been configured with a bean resolver it is possible to look up beans from an expression using the @ symbol.

ExpressionParser parser = new SpelExpressionParser();

StandardEvaluationContext context = new StandardEvaluationContext();

context.setBeanResolver(new MyBeanResolver());

// This will end up calling resolve(context,"foo") on MyBeanResolver during evaluation

Object bean = parser.parseExpression("@foo").getValue(context);

To access a factory bean itself, the bean name should instead be prefixed with an & symbol.

ExpressionParser parser = new SpelExpressionParser();

StandardEvaluationContext context = new StandardEvaluationContext();

context.setBeanResolver(new MyBeanResolver());

// This will end up calling resolve(context,"&foo") on MyBeanResolver during evaluation

Object bean = parser.parseExpression("&foo").getValue(context);

4.4.14. Ternary Operator (If-Then-Else)

You can use the ternary operator for performing if-then-else conditional logic inside the expression. A minimal example is:

String falseString = parser.parseExpression(

"false ? 'trueExp' : 'falseExp'").getValue(String.class);

In this case, the boolean false results in returning the string value 'falseExp'. A more realistic example is shown below.

parser.parseExpression("Name").setValue(societyContext, "IEEE");

societyContext.setVariable("queryName", "Nikola Tesla");

expression = "isMember(#queryName)? #queryName + ' is a member of the ' " +

"+ Name + ' Society' : #queryName + ' is not a member of the ' + Name + ' Society'";

String queryResultString = parser.parseExpression(expression)

.getValue(societyContext, String.class);

// queryResultString = "Nikola Tesla is a member of the IEEE Society"

Also see the next section on the Elvis operator for an even shorter syntax for the ternary operator.

4.4.15. The Elvis Operator

The Elvis operator is a shortening of the ternary operator syntax and is used in the Groovy language. With the ternary operator syntax you usually have to repeat a variable twice, for example:

String name = "Elvis Presley";

String displayName = (name != null ? name : "Unknown");

Instead you can use the Elvis operator, named for the resemblance to Elvis' hair style.

ExpressionParser parser = new SpelExpressionParser();

String name = parser.parseExpression("name?:'Unknown'").getValue(String.class);

System.out.println(name); // 'Unknown'

Here is a more complex example.

ExpressionParser parser = new SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");

String name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla, String.class);

System.out.println(name); // Nikola Tesla

tesla.setName(null);

name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context, tesla, String.class);

System.out.println(name); // Elvis Presley

4.4.16. Safe Navigation operator

The Safe Navigation operator is used to avoid a NullPointerException and comes from the Groovy language. Typically when you have a reference to an object you might need to verify that it is not null before accessing methods or properties of the object. To avoid this, the safe navigation operator will simply return null instead of throwing an exception.

ExpressionParser parser = new SpelExpressionParser();

EvaluationContext context = SimpleEvaluationContext.forReadOnlyDataBinding().build();

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");

tesla.setPlaceOfBirth(new PlaceOfBirth("Smiljan"));

String city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, tesla, String.class);

System.out.println(city); // Smiljan

tesla.setPlaceOfBirth(null);

city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, tesla, String.class);

System.out.println(city); // null - does not throw NullPointerException!!!

The Elvis operator can be used to apply default values in expressions, e.g. in an @Value expression:

@Value("#{systemProperties['pop3.port'] ?: 25}")

This will inject a system property pop3.port if it is defined or 25 if not.

4.4.17. Collection Selection

Selection is a powerful expression language feature that allows you to transform some source collection into another by selecting from its entries.

Selection uses the syntax .?[selectionExpression]. This will filter the collection and return a new collection containing a subset of the original elements. For example, selection would allow us to easily get a list of Serbian inventors:

List<Inventor> list = (List<Inventor>) parser.parseExpression(

"Members.?[Nationality == 'Serbian']").getValue(societyContext);

Selection is possible upon both lists and maps. In the former case the selection criteria is evaluated against each individual list element whilst against a map the selection criteria is evaluated against each map entry (objects of the Java type Map.Entry). Map entries have their key and value accessible as properties for use in the selection.

This expression will return a new map consisting of those elements of the original map where the entry value is less than 27.

Map newMap = parser.parseExpression("map.?[value<27]").getValue();

In addition to returning all the selected elements, it is possible to retrieve just the first or the last value. To obtain the first entry matching the selection the syntax is .^[selectionExpression] whilst to obtain the last matching selection the syntax is .$[selectionExpression].

4.4.18. Collection Projection

Projection allows a collection to drive the evaluation of a sub-expression and the result is a new collection. The syntax for projection is .![projectionExpression]. Most easily understood by example, suppose we have a list of inventors but want the list of cities where they were born. Effectively we want to evaluate 'placeOfBirth.city' for every entry in the inventor list. Using projection:

// returns ['Smiljan', 'Idvor' ]

List placesOfBirth = (List)parser.parseExpression("Members.![placeOfBirth.city]");

A map can also be used to drive projection and in this case the projection expression is evaluated against each entry in the map (represented as a Java Map.Entry). The result of a projection across a map is a list consisting of the evaluation of the projection expression against each map entry.

4.4.19. Expression templating

Expression templates allow a mixing of literal text with one or more evaluation blocks. Each evaluation block is delimited with prefix and suffix characters that you can define, a common choice is to use #{ } as the delimiters. For example,

String randomPhrase = parser.parseExpression(

"random number is #{T(java.lang.Math).random()}",

new TemplateParserContext()).getValue(String.class);

// evaluates to "random number is 0.7038186818312008"

The string is evaluated by concatenating the literal text 'random number is ' with the result of evaluating the expression inside the #{ } delimiter, in this case the result of calling that random() method. The second argument to the method parseExpression() is of the type ParserContext. The ParserContext interface is used to influence how the expression is parsed in order to support the expression templating functionality. The definition of TemplateParserContext is shown below.

public class TemplateParserContext implements ParserContext {

public String getExpressionPrefix() {

return "#{";

}

public String getExpressionSuffix() {

return "}";

}

public boolean isTemplate() {

return true;

}

}

4.5. Classes used in the examples

Inventor.java

package org.spring.samples.spel.inventor;

import java.util.Date;

import java.util.GregorianCalendar;

public class Inventor {

private String name;

private String nationality;

private String[] inventions;

private Date birthdate;

private PlaceOfBirth placeOfBirth;

public Inventor(String name, String nationality) {

GregorianCalendar c= new GregorianCalendar();

this.name = name;

this.nationality = nationality;

this.birthdate = c.getTime();

}

public Inventor(String name, Date birthdate, String nationality) {

this.name = name;

this.nationality = nationality;

this.birthdate = birthdate;

}

public Inventor() {

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getNationality() {

return nationality;

}

public void setNationality(String nationality) {

this.nationality = nationality;

}

public Date getBirthdate() {

return birthdate;

}

public void setBirthdate(Date birthdate) {

this.birthdate = birthdate;

}

public PlaceOfBirth getPlaceOfBirth() {

return placeOfBirth;

}

public void setPlaceOfBirth(PlaceOfBirth placeOfBirth) {

this.placeOfBirth = placeOfBirth;

}

public void setInventions(String[] inventions) {

this.inventions = inventions;

}

public String[] getInventions() {

return inventions;

}

}

PlaceOfBirth.java

package org.spring.samples.spel.inventor;

public class PlaceOfBirth {

private String city;

private String country;

public PlaceOfBirth(String city) {

this.city=city;

}

public PlaceOfBirth(String city, String country) {

this(city);

this.country = country;

}

public String getCity() {

return city;

}

public void setCity(String s) {

this.city = s;

}

public String getCountry() {

return country;

}

public void setCountry(String country) {

this.country = country;

}

}

Society.java

package org.spring.samples.spel.inventor;

import java.util.\*;

public class Society {

private String name;

public static String Advisors = "advisors";

public static String President = "president";

private List<Inventor> members = new ArrayList<Inventor>();

private Map officers = new HashMap();

public List getMembers() {

return members;

}

public Map getOfficers() {

return officers;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public boolean isMember(String name) {

for (Inventor inventor : members) {

if (inventor.getName().equals(name)) {

return true;

}

}

return false;

}

}

5. Aspect Oriented Programming with Spring

5.1. Introduction

Aspect-Oriented Programming (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the aspect. Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects. (Such concerns are often termed crosscutting concerns in AOP literature.)

One of the key components of Spring is the AOP framework. While the Spring IoC container does not depend on AOP, meaning you do not need to use AOP if you don’t want to, AOP complements Spring IoC to provide a very capable middleware solution.

Spring 2.0+ AOP

Spring 2.0 introduced a simpler and more powerful way of writing custom aspects using either a schema-based approach or the @AspectJ annotation style. Both of these styles offer fully typed advice and use of the AspectJ pointcut language, while still using Spring AOP for weaving.

The Spring 2.0+ schema- and @AspectJ-based AOP support is discussed in this chapter. The lower-level AOP support, as commonly exposed in Spring 1.2 applications, is discussed in the following chapter.

AOP is used in the Spring Framework to…​

…​ provide declarative enterprise services, especially as a replacement for EJB declarative services. The most important such service is declarative transaction management.

…​ allow users to implement custom aspects, complementing their use of OOP with AOP.

If you are interested only in generic declarative services or other pre-packaged declarative middleware services such as pooling, you do not need to work directly with Spring AOP, and can skip most of this chapter.

5.1.1. AOP concepts

Let us begin by defining some central AOP concepts and terminology. These terms are not Spring-specific…​ unfortunately, AOP terminology is not particularly intuitive; however, it would be even more confusing if Spring used its own terminology.

Aspect: a modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the @Aspect annotation (the @AspectJ style).

Join point: a point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.

Advice: action taken by an aspect at a particular join point. Different types of advice include "around", "before" and "after" advice. (Advice types are discussed below.) Many AOP frameworks, including Spring, model an advice as an interceptor, maintaining a chain of interceptors around the join point.

Pointcut: a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP, and Spring uses the AspectJ pointcut expression language by default.

Introduction: declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any advised object. For example, you could use an introduction to make a bean implement an IsModified interface, to simplify caching. (An introduction is known as an inter-type declaration in the AspectJ community.)

Target object: object being advised by one or more aspects. Also referred to as the advised object. Since Spring AOP is implemented using runtime proxies, this object will always be a proxied object.

AOP proxy: an object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.

Weaving: linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of advice:

Before advice: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

After returning advice: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.

After throwing advice: Advice to be executed if a method exits by throwing an exception.

After (finally) advice: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).

Around advice: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Around advice is the most general kind of advice. Since Spring AOP, like AspectJ, provides a full range of advice types, we recommend that you use the least powerful advice type that can implement the required behavior. For example, if you need only to update a cache with the return value of a method, you are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you do not need to invoke the proceed() method on the JoinPoint used for around advice, and hence cannot fail to invoke it.

In Spring 2.0, all advice parameters are statically typed, so that you work with advice parameters of the appropriate type (the type of the return value from a method execution for example) rather than Object arrays.

The concept of join points, matched by pointcuts, is the key to AOP which distinguishes it from older technologies offering only interception. Pointcuts enable advice to be targeted independently of the Object-Oriented hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects (such as all business operations in the service layer).

5.1.2. Spring AOP capabilities and goals

Spring AOP is implemented in pure Java. There is no need for a special compilation process. Spring AOP does not need to control the class loader hierarchy, and is thus suitable for use in a Servlet container or application server.

Spring AOP currently supports only method execution join points (advising the execution of methods on Spring beans). Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs. If you need to advise field access and update join points, consider a language such as AspectJ.

Spring AOP’s approach to AOP differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable); it is rather to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications.

Thus, for example, the Spring Framework’s AOP functionality is normally used in conjunction with the Spring IoC container. Aspects are configured using normal bean definition syntax (although this allows powerful "autoproxying" capabilities): this is a crucial difference from other AOP implementations. There are some things you cannot do easily or efficiently with Spring AOP, such as advise very fine-grained objects (such as domain objects typically): AspectJ is the best choice in such cases. However, our experience is that Spring AOP provides an excellent solution to most problems in enterprise Java applications that are amenable to AOP.

Spring AOP will never strive to compete with AspectJ to provide a comprehensive AOP solution. We believe that both proxy-based frameworks like Spring AOP and full-blown frameworks such as AspectJ are valuable, and that they are complementary, rather than in competition. Spring seamlessly integrates Spring AOP and IoC with AspectJ, to enable all uses of AOP to be catered for within a consistent Spring-based application architecture. This integration does not affect the Spring AOP API or the AOP Alliance API: Spring AOP remains backward-compatible. See the following chapter for a discussion of the Spring AOP APIs.

One of the central tenets of the Spring Framework is that of non-invasiveness; this is the idea that you should not be forced to introduce framework-specific classes and interfaces into your business/domain model. However, in some places the Spring Framework does give you the option to introduce Spring Framework-specific dependencies into your codebase: the rationale in giving you such options is because in certain scenarios it might be just plain easier to read or code some specific piece of functionality in such a way. The Spring Framework (almost) always offers you the choice though: you have the freedom to make an informed decision as to which option best suits your particular use case or scenario.

One such choice that is relevant to this chapter is that of which AOP framework (and which AOP style) to choose. You have the choice of AspectJ and/or Spring AOP, and you also have the choice of either the @AspectJ annotation-style approach or the Spring XML configuration-style approach. The fact that this chapter chooses to introduce the @AspectJ-style approach first should not be taken as an indication that the Spring team favors the @AspectJ annotation-style approach over the Spring XML configuration-style.

See Choosing which AOP declaration style to use for a more complete discussion of the whys and wherefores of each style.

5.1.3. AOP Proxies

Spring AOP defaults to using standard JDK dynamic proxies for AOP proxies. This enables any interface (or set of interfaces) to be proxied.

Spring AOP can also use CGLIB proxies. This is necessary to proxy classes rather than interfaces. CGLIB is used by default if a business object does not implement an interface. As it is good practice to program to interfaces rather than classes; business classes normally will implement one or more business interfaces. It is possible to force the use of CGLIB, in those (hopefully rare) cases where you need to advise a method that is not declared on an interface, or where you need to pass a proxied object to a method as a concrete type.

It is important to grasp the fact that Spring AOP is proxy-based. See Understanding AOP proxies for a thorough examination of exactly what this implementation detail actually means.

5.2. @AspectJ support

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with annotations. The @AspectJ style was introduced by the AspectJ project as part of the AspectJ 5 release. Spring interprets the same annotations as AspectJ 5, using a library supplied by AspectJ for pointcut parsing and matching. The AOP runtime is still pure Spring AOP though, and there is no dependency on the AspectJ compiler or weaver.

Using the AspectJ compiler and weaver enables use of the full AspectJ language, and is discussed in Using AspectJ with Spring applications.

5.2.1. Enabling @AspectJ Support

To use @AspectJ aspects in a Spring configuration you need to enable Spring support for configuring Spring AOP based on @AspectJ aspects, and autoproxying beans based on whether or not they are advised by those aspects. By autoproxying we mean that if Spring determines that a bean is advised by one or more aspects, it will automatically generate a proxy for that bean to intercept method invocations and ensure that advice is executed as needed.

The @AspectJ support can be enabled with XML or Java style configuration. In either case you will also need to ensure that AspectJ’s aspectjweaver.jar library is on the classpath of your application (version 1.8 or later). This library is available in the 'lib' directory of an AspectJ distribution or via the Maven Central repository.

Enabling @AspectJ Support with Java configuration

To enable @AspectJ support with Java @Configuration add the @EnableAspectJAutoProxy annotation:

@Configuration

@EnableAspectJAutoProxy

public class AppConfig {

}

Enabling @AspectJ Support with XML configuration

To enable @AspectJ support with XML based configuration use the aop:aspectj-autoproxy element:

<aop:aspectj-autoproxy/>

This assumes that you are using schema support as described in XML Schema-based configuration. See the AOP schema for how to import the tags in the aop namespace.

5.2.2. Declaring an aspect

With the @AspectJ support enabled, any bean defined in your application context with a class that is an @AspectJ aspect (has the @Aspect annotation) will be automatically detected by Spring and used to configure Spring AOP. The following example shows the minimal definition required for a not-very-useful aspect:

A regular bean definition in the application context, pointing to a bean class that has the @Aspect annotation:

<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">

<!-- configure properties of aspect here as normal -->

</bean>

And the NotVeryUsefulAspect class definition, annotated with org.aspectj.lang.annotation.Aspect annotation;

package org.xyz;

import org.aspectj.lang.annotation.Aspect;

@Aspect

public class NotVeryUsefulAspect {

}

Aspects (classes annotated with @Aspect) may have methods and fields just like any other class. They may also contain pointcut, advice, and introduction (inter-type) declarations.

Autodetecting aspects through component scanning

You may register aspect classes as regular beans in your Spring XML configuration, or autodetect them through classpath scanning - just like any other Spring-managed bean. However, note that the @Aspect annotation is not sufficient for autodetection in the classpath: For that purpose, you need to add a separate @Component annotation (or alternatively a custom stereotype annotation that qualifies, as per the rules of Spring’s component scanner).

Advising aspects with other aspects?

In Spring AOP, it is not possible to have aspects themselves be the target of advice from other aspects. The @Aspect annotation on a class marks it as an aspect, and hence excludes it from auto-proxying.

5.2.3. Declaring a pointcut

Recall that pointcuts determine join points of interest, and thus enable us to control when advice executes. Spring AOP only supports method execution join points for Spring beans, so you can think of a pointcut as matching the execution of methods on Spring beans. A pointcut declaration has two parts: a signature comprising a name and any parameters, and a pointcut expression that determines exactly which method executions we are interested in. In the @AspectJ annotation-style of AOP, a pointcut signature is provided by a regular method definition, and the pointcut expression is indicated using the @Pointcut annotation (the method serving as the pointcut signature must have a void return type).

An example will help make this distinction between a pointcut signature and a pointcut expression clear. The following example defines a pointcut named 'anyOldTransfer' that will match the execution of any method named 'transfer':

@Pointcut("execution(\* transfer(..))")// the pointcut expression

private void anyOldTransfer() {}// the pointcut signature

The pointcut expression that forms the value of the @Pointcut annotation is a regular AspectJ 5 pointcut expression. For a full discussion of AspectJ’s pointcut language, see the AspectJ Programming Guide (and for extensions, the AspectJ 5 Developers Notebook) or one of the books on AspectJ such as "Eclipse AspectJ" by Colyer et. al. or "AspectJ in Action" by Ramnivas Laddad.

Supported Pointcut Designators

Spring AOP supports the following AspectJ pointcut designators (PCD) for use in pointcut expressions:

Other pointcut types

The full AspectJ pointcut language supports additional pointcut designators that are not supported in Spring. These are: call, get, set, preinitialization, staticinitialization, initialization, handler, adviceexecution, withincode, cflow, cflowbelow, if, @this, and @withincode. Use of these pointcut designators in pointcut expressions interpreted by Spring AOP will result in an IllegalArgumentException being thrown.

The set of pointcut designators supported by Spring AOP may be extended in future releases to support more of the AspectJ pointcut designators.

execution - for matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP

within - limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)

this - limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type

target - limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type

args - limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types

@target - limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type

@args - limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given type(s)

@within - limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)

@annotation - limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation

Because Spring AOP limits matching to only method execution join points, the discussion of the pointcut designators above gives a narrower definition than you will find in the AspectJ programming guide. In addition, AspectJ itself has type-based semantics and at an execution join point both this and target refer to the same object - the object executing the method. Spring AOP is a proxy-based system and differentiates between the proxy object itself (bound to this) and the target object behind the proxy (bound to target).

Due to the proxy-based nature of Spring’s AOP framework, calls within the target object are by definition not intercepted. For JDK proxies, only public interface method calls on the proxy can be intercepted. With CGLIB, public and protected method calls on the proxy will be intercepted, and even package-visible methods if necessary. However, common interactions through proxies should always be designed through public signatures.

Note that pointcut definitions are generally matched against any intercepted method. If a pointcut is strictly meant to be public-only, even in a CGLIB proxy scenario with potential non-public interactions through proxies, it needs to be defined accordingly.

If your interception needs include method calls or even constructors within the target class, consider the use of Spring-driven native AspectJ weaving instead of Spring’s proxy-based AOP framework. This constitutes a different mode of AOP usage with different characteristics, so be sure to make yourself familiar with weaving first before making a decision.

Spring AOP also supports an additional PCD named bean. This PCD allows you to limit the matching of join points to a particular named Spring bean, or to a set of named Spring beans (when using wildcards). The bean PCD has the following form:

bean(idOrNameOfBean)

The idOrNameOfBean token can be the name of any Spring bean: limited wildcard support using the \* character is provided, so if you establish some naming conventions for your Spring beans you can quite easily write a bean PCD expression to pick them out. As is the case with other pointcut designators, the bean PCD can be &&'ed, ||'ed, and ! (negated) too.

Please note that the bean PCD is only supported in Spring AOP - and not in native AspectJ weaving. It is a Spring-specific extension to the standard PCDs that AspectJ defines and therefore not available for aspects declared in the @Aspect model.

The bean PCD operates at the instance level (building on the Spring bean name concept) rather than at the type level only (which is what weaving-based AOP is limited to). Instance-based pointcut designators are a special capability of Spring’s proxy-based AOP framework and its close integration with the Spring bean factory, where it is natural and straightforward to identify specific beans by name.

Combining pointcut expressions

Pointcut expressions can be combined using '&&', '||' and '!'. It is also possible to refer to pointcut expressions by name. The following example shows three pointcut expressions: anyPublicOperation (which matches if a method execution join point represents the execution of any public method); inTrading (which matches if a method execution is in the trading module), and tradingOperation (which matches if a method execution represents any public method in the trading module).

@Pointcut("execution(public \* \*(..))")

private void anyPublicOperation() {}

@Pointcut("within(com.xyz.someapp.trading..\*)")

private void inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()")

private void tradingOperation() {}

It is a best practice to build more complex pointcut expressions out of smaller named components as shown above. When referring to pointcuts by name, normal Java visibility rules apply (you can see private pointcuts in the same type, protected pointcuts in the hierarchy, public pointcuts anywhere and so on). Visibility does not affect pointcut matching.

Sharing common pointcut definitions

When working with enterprise applications, you often want to refer to modules of the application and particular sets of operations from within several aspects. We recommend defining a "SystemArchitecture" aspect that captures common pointcut expressions for this purpose. A typical such aspect would look as follows:

package com.xyz.someapp;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Pointcut;

@Aspect

public class SystemArchitecture {

/\*\*

\* A join point is in the web layer if the method is defined

\* in a type in the com.xyz.someapp.web package or any sub-package

\* under that.

\*/

@Pointcut("within(com.xyz.someapp.web..\*)")

public void inWebLayer() {}

/\*\*

\* A join point is in the service layer if the method is defined

\* in a type in the com.xyz.someapp.service package or any sub-package

\* under that.

\*/

@Pointcut("within(com.xyz.someapp.service..\*)")

public void inServiceLayer() {}

/\*\*

\* A join point is in the data access layer if the method is defined

\* in a type in the com.xyz.someapp.dao package or any sub-package

\* under that.

\*/

@Pointcut("within(com.xyz.someapp.dao..\*)")

public void inDataAccessLayer() {}

/\*\*

\* A business service is the execution of any method defined on a service

\* interface. This definition assumes that interfaces are placed in the

\* "service" package, and that implementation types are in sub-packages.

\*

\* If you group service interfaces by functional area (for example,

\* in packages com.xyz.someapp.abc.service and com.xyz.someapp.def.service) then

\* the pointcut expression "execution(\* com.xyz.someapp..service.\*.\*(..))"

\* could be used instead.

\*

\* Alternatively, you can write the expression using the 'bean'

\* PCD, like so "bean(\*Service)". (This assumes that you have

\* named your Spring service beans in a consistent fashion.)

\*/

@Pointcut("execution(\* com.xyz.someapp..service.\*.\*(..))")

public void businessService() {}

/\*\*

\* A data access operation is the execution of any method defined on a

\* dao interface. This definition assumes that interfaces are placed in the

\* "dao" package, and that implementation types are in sub-packages.

\*/

@Pointcut("execution(\* com.xyz.someapp.dao.\*.\*(..))")

public void dataAccessOperation() {}

}

The pointcuts defined in such an aspect can be referred to anywhere that you need a pointcut expression. For example, to make the service layer transactional, you could write:

<aop:config>

<aop:advisor

pointcut="com.xyz.someapp.SystemArchitecture.businessService()"

advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">

<tx:attributes>

<tx:method name="\*" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

The <aop:config> and <aop:advisor> elements are discussed in Schema-based AOP support. The transaction elements are discussed in Transaction Management.

Examples

Spring AOP users are likely to use the execution pointcut designator the most often. The format of an execution expression is:

execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-pattern(param-pattern)

throws-pattern?)

All parts except the returning type pattern (ret-type-pattern in the snippet above), name pattern, and parameters pattern are optional. The returning type pattern determines what the return type of the method must be in order for a join point to be matched. Most frequently you will use \* as the returning type pattern, which matches any return type. A fully-qualified type name will match only when the method returns the given type. The name pattern matches the method name. You can use the \* wildcard as all or part of a name pattern. If specifying a declaring type pattern then include a trailing . to join it to the name pattern component. The parameters pattern is slightly more complex: () matches a method that takes no parameters, whereas (..) matches any number of parameters (zero or more). The pattern (\*) matches a method taking one parameter of any type, (\*,String) matches a method taking two parameters, the first can be of any type, the second must be a String. Consult the Language Semantics section of the AspectJ Programming Guide for more information.

Some examples of common pointcut expressions are given below.

the execution of any public method:

execution(public \* \*(..))

the execution of any method with a name beginning with "set":

execution(\* set\*(..))

the execution of any method defined by the AccountService interface:

execution(\* com.xyz.service.AccountService.\*(..))

the execution of any method defined in the service package:

execution(\* com.xyz.service.\*.\*(..))

the execution of any method defined in the service package or a sub-package:

execution(\* com.xyz.service..\*.\*(..))

any join point (method execution only in Spring AOP) within the service package:

within(com.xyz.service.\*)

any join point (method execution only in Spring AOP) within the service package or a sub-package:

within(com.xyz.service..\*)

any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface:

this(com.xyz.service.AccountService)

'this' is more commonly used in a binding form :- see the following section on advice for how to make the proxy object available in the advice body.

any join point (method execution only in Spring AOP) where the target object implements the AccountService interface:

target(com.xyz.service.AccountService)

'target' is more commonly used in a binding form :- see the following section on advice for how to make the target object available in the advice body.

any join point (method execution only in Spring AOP) which takes a single parameter, and where the argument passed at runtime is Serializable:

args(java.io.Serializable)

'args' is more commonly used in a binding form :- see the following section on advice for how to make the method arguments available in the advice body.

Note that the pointcut given in this example is different to execution(\* \*(java.io.Serializable)): the args version matches if the argument passed at runtime is Serializable, the execution version matches if the method signature declares a single parameter of type Serializable.

any join point (method execution only in Spring AOP) where the target object has an @Transactional annotation:

@target(org.springframework.transaction.annotation.Transactional)

'@target' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

any join point (method execution only in Spring AOP) where the declared type of the target object has an @Transactional annotation:

@within(org.springframework.transaction.annotation.Transactional)

'@within' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

any join point (method execution only in Spring AOP) where the executing method has an @Transactional annotation:

@annotation(org.springframework.transaction.annotation.Transactional)

'@annotation' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the @Classified annotation:

@args(com.xyz.security.Classified)

'@args' can also be used in a binding form :- see the following section on advice for how to make the annotation object(s) available in the advice body.

any join point (method execution only in Spring AOP) on a Spring bean named tradeService:

bean(tradeService)

any join point (method execution only in Spring AOP) on Spring beans having names that match the wildcard expression \*Service:

bean(\*Service)

Writing good pointcuts

During compilation, AspectJ processes pointcuts in order to try and optimize matching performance. Examining code and determining if each join point matches (statically or dynamically) a given pointcut is a costly process. (A dynamic match means the match cannot be fully determined from static analysis and a test will be placed in the code to determine if there is an actual match when the code is running). On first encountering a pointcut declaration, AspectJ will rewrite it into an optimal form for the matching process. What does this mean? Basically pointcuts are rewritten in DNF (Disjunctive Normal Form) and the components of the pointcut are sorted such that those components that are cheaper to evaluate are checked first. This means you do not have to worry about understanding the performance of various pointcut designators and may supply them in any order in a pointcut declaration.

However, AspectJ can only work with what it is told, and for optimal performance of matching you should think about what they are trying to achieve and narrow the search space for matches as much as possible in the definition. The existing designators naturally fall into one of three groups: kinded, scoping and context:

Kinded designators are those which select a particular kind of join point. For example: execution, get, set, call, handler

Scoping designators are those which select a group of join points of interest (of probably many kinds). For example: within, withincode

Contextual designators are those that match (and optionally bind) based on context. For example: this, target, @annotation

A well written pointcut should try and include at least the first two types (kinded and scoping), whilst the contextual designators may be included if wishing to match based on join point context, or bind that context for use in the advice. Supplying either just a kinded designator or just a contextual designator will work but could affect weaving performance (time and memory used) due to all the extra processing and analysis. Scoping designators are very fast to match and their usage means AspectJ can very quickly dismiss groups of join points that should not be further processed - that is why a good pointcut should always include one if possible.

5.2.4. Declaring advice

Advice is associated with a pointcut expression, and runs before, after, or around method executions matched by the pointcut. The pointcut expression may be either a simple reference to a named pointcut, or a pointcut expression declared in place.

Before advice

Before advice is declared in an aspect using the @Before annotation:

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

@Aspect

public class BeforeExample {

@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

public void doAccessCheck() {

// ...

}

}

If using an in-place pointcut expression we could rewrite the above example as:

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Before;

@Aspect

public class BeforeExample {

@Before("execution(\* com.xyz.myapp.dao.\*.\*(..))")

public void doAccessCheck() {

// ...

}

}

After returning advice

After returning advice runs when a matched method execution returns normally. It is declared using the @AfterReturning annotation:

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.AfterReturning;

@Aspect

public class AfterReturningExample {

@AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

public void doAccessCheck() {

// ...

}

}

Note: it is of course possible to have multiple advice declarations, and other members as well, all inside the same aspect. We’re just showing a single advice declaration in these examples to focus on the issue under discussion at the time.

Sometimes you need access in the advice body to the actual value that was returned. You can use the form of @AfterReturning that binds the return value for this:

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.AfterReturning;

@Aspect

public class AfterReturningExample {

@AfterReturning(

pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",

returning="retVal")

public void doAccessCheck(Object retVal) {

// ...

}

}

The name used in the returning attribute must correspond to the name of a parameter in the advice method. When a method execution returns, the return value will be passed to the advice method as the corresponding argument value. A returning clause also restricts matching to only those method executions that return a value of the specified type ( Object in this case, which will match any return value).

Please note that it is not possible to return a totally different reference when using after-returning advice.

After throwing advice

After throwing advice runs when a matched method execution exits by throwing an exception. It is declared using the @AfterThrowing annotation:

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.AfterThrowing;

@Aspect

public class AfterThrowingExample {

@AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

public void doRecoveryActions() {

// ...

}

}

Often you want the advice to run only when exceptions of a given type are thrown, and you also often need access to the thrown exception in the advice body. Use the throwing attribute to both restrict matching (if desired, use Throwable as the exception type otherwise) and bind the thrown exception to an advice parameter.

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.AfterThrowing;

@Aspect

public class AfterThrowingExample {

@AfterThrowing(

pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",

throwing="ex")

public void doRecoveryActions(DataAccessException ex) {

// ...

}

}

The name used in the throwing attribute must correspond to the name of a parameter in the advice method. When a method execution exits by throwing an exception, the exception will be passed to the advice method as the corresponding argument value. A throwing clause also restricts matching to only those method executions that throw an exception of the specified type ( DataAccessException in this case).

After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the @After annotation. After advice must be prepared to handle both normal and exception return conditions. It is typically used for releasing resources, etc.

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.After;

@Aspect

public class AfterFinallyExample {

@After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")

public void doReleaseLock() {

// ...

}

}

Around advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements (i.e. don’t use around advice if simple before advice would do).

Around advice is declared using the @Around annotation. The first parameter of the advice method must be of type ProceedingJoinPoint. Within the body of the advice, calling proceed() on the ProceedingJoinPoint causes the underlying method to execute. The proceed method may also be called passing in an Object[] - the values in the array will be used as the arguments to the method execution when it proceeds.

The behavior of proceed when called with an Object[] is a little different than the behavior of proceed for around advice compiled by the AspectJ compiler. For around advice written using the traditional AspectJ language, the number of arguments passed to proceed must match the number of arguments passed to the around advice (not the number of arguments taken by the underlying join point), and the value passed to proceed in a given argument position supplants the original value at the join point for the entity the value was bound to (Don’t worry if this doesn’t make sense right now!). The approach taken by Spring is simpler and a better match to its proxy-based, execution only semantics. You only need to be aware of this difference if you are compiling @AspectJ aspects written for Spring and using proceed with arguments with the AspectJ compiler and weaver. There is a way to write such aspects that is 100% compatible across both Spring AOP and AspectJ, and this is discussed in the following section on advice parameters.

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.ProceedingJoinPoint;

@Aspect

public class AroundExample {

@Around("com.xyz.myapp.SystemArchitecture.businessService()")

public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {

// start stopwatch

Object retVal = pjp.proceed();

// stop stopwatch

return retVal;

}

}

The value returned by the around advice will be the return value seen by the caller of the method. A simple caching aspect for example could return a value from a cache if it has one, and invoke proceed() if it does not. Note that proceed may be invoked once, many times, or not at all within the body of the around advice, all of these are quite legal.

Advice parameters

Spring offers fully typed advice - meaning that you declare the parameters you need in the advice signature (as we saw for the returning and throwing examples above) rather than work with Object[] arrays all the time. We’ll see how to make argument and other contextual values available to the advice body in a moment. First let’s take a look at how to write generic advice that can find out about the method the advice is currently advising.

Access to the current JoinPoint

Any advice method may declare as its first parameter, a parameter of type org.aspectj.lang.JoinPoint (please note that around advice is required to declare a first parameter of type ProceedingJoinPoint, which is a subclass of JoinPoint. The JoinPoint interface provides a number of useful methods such as getArgs() (returns the method arguments), getThis() (returns the proxy object), getTarget() (returns the target object), getSignature() (returns a description of the method that is being advised) and toString() (prints a useful description of the method being advised). Please do consult the javadocs for full details.

Passing parameters to advice

We’ve already seen how to bind the returned value or exception value (using after returning and after throwing advice). To make argument values available to the advice body, you can use the binding form of args. If a parameter name is used in place of a type name in an args expression, then the value of the corresponding argument will be passed as the parameter value when the advice is invoked. An example should make this clearer. Suppose you want to advise the execution of dao operations that take an Account object as the first parameter, and you need access to the account in the advice body. You could write the following:

@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,..)")

public void validateAccount(Account account) {

// ...

}

The args(account,..) part of the pointcut expression serves two purposes: firstly, it restricts matching to only those method executions where the method takes at least one parameter, and the argument passed to that parameter is an instance of Account; secondly, it makes the actual Account object available to the advice via the account parameter.

Another way of writing this is to declare a pointcut that "provides" the Account object value when it matches a join point, and then just refer to the named pointcut from the advice. This would look as follows:

@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,..)")

private void accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")

public void validateAccount(Account account) {

// ...

}

The interested reader is once more referred to the AspectJ programming guide for more details.

The proxy object ( this), target object ( target), and annotations ( @within, @target, @annotation, @args) can all be bound in a similar fashion. The following example shows how you could match the execution of methods annotated with an @Auditable annotation, and extract the audit code.

First the definition of the @Auditable annotation:

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

public @interface Auditable {

AuditCode value();

}

And then the advice that matches the execution of @Auditable methods:

@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)")

public void audit(Auditable auditable) {

AuditCode code = auditable.value();

// ...

}

Advice parameters and generics

Spring AOP can handle generics used in class declarations and method parameters. Suppose you have a generic type like this:

public interface Sample<T> {

void sampleGenericMethod(T param);

void sampleGenericCollectionMethod(Collection<T> param);

}

You can restrict interception of method types to certain parameter types by simply typing the advice parameter to the parameter type you want to intercept the method for:

@Before("execution(\* ..Sample+.sampleGenericMethod(\*)) && args(param)")

public void beforeSampleMethod(MyType param) {

// Advice implementation

}

That this works is pretty obvious as we already discussed above. However, it’s worth pointing out that this won’t work for generic collections. So you cannot define a pointcut like this:

@Before("execution(\* ..Sample+.sampleGenericCollectionMethod(\*)) && args(param)")

public void beforeSampleMethod(Collection<MyType> param) {

// Advice implementation

}

To make this work we would have to inspect every element of the collection, which is not reasonable as we also cannot decide how to treat null values in general. To achieve something similar to this you have to type the parameter to Collection<?> and manually check the type of the elements.

Determining argument names

The parameter binding in advice invocations relies on matching names used in pointcut expressions to declared parameter names in (advice and pointcut) method signatures. Parameter names are not available through Java reflection, so Spring AOP uses the following strategies to determine parameter names:

If the parameter names have been specified by the user explicitly, then the specified parameter names are used: both the advice and the pointcut annotations have an optional "argNames" attribute which can be used to specify the argument names of the annotated method - these argument names are available at runtime. For example:

@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",

argNames="bean,auditable")

public void audit(Object bean, Auditable auditable) {

AuditCode code = auditable.value();

// ... use code and bean

}

If the first parameter is of the JoinPoint, ProceedingJoinPoint, or JoinPoint.StaticPart type, you may leave out the name of the parameter from the value of the "argNames" attribute. For example, if you modify the preceding advice to receive the join point object, the "argNames" attribute need not include it:

@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",

argNames="bean,auditable")

public void audit(JoinPoint jp, Object bean, Auditable auditable) {

AuditCode code = auditable.value();

// ... use code, bean, and jp

}

The special treatment given to the first parameter of the JoinPoint, ProceedingJoinPoint, and JoinPoint.StaticPart types is particularly convenient for advice that do not collect any other join point context. In such situations, you may simply omit the "argNames" attribute. For example, the following advice need not declare the "argNames" attribute:

@Before("com.xyz.lib.Pointcuts.anyPublicMethod()")

public void audit(JoinPoint jp) {

// ... use jp

}

Using the 'argNames' attribute is a little clumsy, so if the 'argNames' attribute has not been specified, then Spring AOP will look at the debug information for the class and try to determine the parameter names from the local variable table. This information will be present as long as the classes have been compiled with debug information ( '-g:vars' at a minimum). The consequences of compiling with this flag on are: (1) your code will be slightly easier to understand (reverse engineer), (2) the class file sizes will be very slightly bigger (typically inconsequential), (3) the optimization to remove unused local variables will not be applied by your compiler. In other words, you should encounter no difficulties building with this flag on.

If an @AspectJ aspect has been compiled by the AspectJ compiler (ajc) even without the debug information then there is no need to add the argNames attribute as the compiler will retain the needed information.

If the code has been compiled without the necessary debug information, then Spring AOP will attempt to deduce the pairing of binding variables to parameters (for example, if only one variable is bound in the pointcut expression, and the advice method only takes one parameter, the pairing is obvious!). If the binding of variables is ambiguous given the available information, then an AmbiguousBindingException will be thrown.

If all of the above strategies fail then an IllegalArgumentException will be thrown.

Proceeding with arguments

We remarked earlier that we would describe how to write a proceed call with arguments that works consistently across Spring AOP and AspectJ. The solution is simply to ensure that the advice signature binds each of the method parameters in order. For example:

@Around("execution(List<Account> find\*(..)) && " +

"com.xyz.myapp.SystemArchitecture.inDataAccessLayer() && " +

"args(accountHolderNamePattern)")

public Object preProcessQueryPattern(ProceedingJoinPoint pjp,

String accountHolderNamePattern) throws Throwable {

String newPattern = preProcess(accountHolderNamePattern);

return pjp.proceed(new Object[] {newPattern});

}

In many cases you will be doing this binding anyway (as in the example above).

Advice ordering

What happens when multiple pieces of advice all want to run at the same join point? Spring AOP follows the same precedence rules as AspectJ to determine the order of advice execution. The highest precedence advice runs first "on the way in" (so given two pieces of before advice, the one with highest precedence runs first). "On the way out" from a join point, the highest precedence advice runs last (so given two pieces of after advice, the one with the highest precedence will run second).

When two pieces of advice defined in different aspects both need to run at the same join point, unless you specify otherwise the order of execution is undefined. You can control the order of execution by specifying precedence. This is done in the normal Spring way by either implementing the org.springframework.core.Ordered interface in the aspect class or annotating it with the Order annotation. Given two aspects, the aspect returning the lower value from Ordered.getValue() (or the annotation value) has the higher precedence.

When two pieces of advice defined in the same aspect both need to run at the same join point, the ordering is undefined (since there is no way to retrieve the declaration order via reflection for javac-compiled classes). Consider collapsing such advice methods into one advice method per join point in each aspect class, or refactor the pieces of advice into separate aspect classes - which can be ordered at the aspect level.

5.2.5. Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

An introduction is made using the @DeclareParents annotation. This annotation is used to declare that matching types have a new parent (hence the name). For example, given an interface UsageTracked, and an implementation of that interface DefaultUsageTracked, the following aspect declares that all implementors of service interfaces also implement the UsageTracked interface. (In order to expose statistics via JMX for example.)

@Aspect

public class UsageTracking {

@DeclareParents(value="com.xzy.myapp.service.\*+", defaultImpl=DefaultUsageTracked.class)

public static UsageTracked mixin;

@Before("com.xyz.myapp.SystemArchitecture.businessService() && this(usageTracked)")

public void recordUsage(UsageTracked usageTracked) {

usageTracked.incrementUseCount();

}

}

The interface to be implemented is determined by the type of the annotated field. The value attribute of the @DeclareParents annotation is an AspectJ type pattern :- any bean of a matching type will implement the UsageTracked interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the UsageTracked interface. If accessing a bean programmatically you would write the following:

UsageTracked usageTracked = (UsageTracked) context.getBean("myService");

5.2.6. Aspect instantiation models

(This is an advanced topic, so if you are just starting out with AOP you can safely skip it until later.)

By default there will be a single instance of each aspect within the application context. AspectJ calls this the singleton instantiation model. It is possible to define aspects with alternate lifecycles :- Spring supports AspectJ’s perthis and pertarget instantiation models ( percflow, percflowbelow, and pertypewithin are not currently supported).

A "perthis" aspect is declared by specifying a perthis clause in the @Aspect annotation. Let’s look at an example, and then we’ll explain how it works.

@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")

public class MyAspect {

private int someState;

@Before(com.xyz.myapp.SystemArchitecture.businessService())

public void recordServiceUsage() {

// ...

}

}

The effect of the 'perthis' clause is that one aspect instance will be created for each unique service object executing a business service (each unique object bound to 'this' at join points matched by the pointcut expression). The aspect instance is created the first time that a method is invoked on the service object. The aspect goes out of scope when the service object goes out of scope. Before the aspect instance is created, none of the advice within it executes. As soon as the aspect instance has been created, the advice declared within it will execute at matched join points, but only when the service object is the one this aspect is associated with. See the AspectJ programming guide for more information on per-clauses.

The 'pertarget' instantiation model works in exactly the same way as perthis, but creates one aspect instance for each unique target object at matched join points.

5.2.7. Example

Now that you have seen how all the constituent parts work, let’s put them together to do something useful!

The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely to succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don’t need to go back to the user for conflict resolution), we’d like to transparently retry the operation to avoid the client seeing a PessimisticLockingFailureException. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.

Because we want to retry the operation, we will need to use around advice so that we can call proceed multiple times. Here’s how the basic aspect implementation looks:

@Aspect

public class ConcurrentOperationExecutor implements Ordered {

private static final int DEFAULT\_MAX\_RETRIES = 2;

private int maxRetries = DEFAULT\_MAX\_RETRIES;

private int order = 1;

public void setMaxRetries(int maxRetries) {

this.maxRetries = maxRetries;

}

public int getOrder() {

return this.order;

}

public void setOrder(int order) {

this.order = order;

}

@Around("com.xyz.myapp.SystemArchitecture.businessService()")

public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {

int numAttempts = 0;

PessimisticLockingFailureException lockFailureException;

do {

numAttempts++;

try {

return pjp.proceed();

}

catch(PessimisticLockingFailureException ex) {

lockFailureException = ex;

}

} while(numAttempts <= this.maxRetries);

throw lockFailureException;

}

}

Note that the aspect implements the Ordered interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The maxRetries and order properties will both be configured by Spring. The main action happens in the doConcurrentOperation around advice. Notice that for the moment we’re applying the retry logic to all businessService()s. We try to proceed, and if we fail with an PessimisticLockingFailureException we simply try again unless we have exhausted all of our retry attempts.

The corresponding Spring configuration is:

<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor" class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">

<property name="maxRetries" value="3"/>

<property name="order" value="100"/>

</bean>

To refine the aspect so that it only retries idempotent operations, we might define an Idempotent annotation:

@Retention(RetentionPolicy.RUNTIME)

public @interface Idempotent {

// marker annotation

}

and use the annotation to annotate the implementation of service operations. The change to the aspect to only retry idempotent operations simply involves refining the pointcut expression so that only @Idempotent operations match:

@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +

"@annotation(com.xyz.myapp.service.Idempotent)")

public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {

...

}

5.3. Schema-based AOP support

If you prefer an XML-based format, then Spring also offers support for defining aspects using the new "aop" namespace tags. The exact same pointcut expressions and advice kinds are supported as when using the @AspectJ style, hence in this section we will focus on the new syntax and refer the reader to the discussion in the previous section (@AspectJ support) for an understanding of writing pointcut expressions and the binding of advice parameters.

To use the aop namespace tags described in this section, you need to import the spring-aop schema as described in XML Schema-based configuration. See the AOP schema for how to import the tags in the aop namespace.

Within your Spring configurations, all aspect and advisor elements must be placed within an <aop:config> element (you can have more than one <aop:config> element in an application context configuration). An <aop:config> element can contain pointcut, advisor, and aspect elements (note these must be declared in that order).

The <aop:config> style of configuration makes heavy use of Spring’s auto-proxying mechanism. This can cause issues (such as advice not being woven) if you are already using explicit auto-proxying via the use of BeanNameAutoProxyCreator or suchlike. The recommended usage pattern is to use either just the <aop:config> style, or just the AutoProxyCreator style.

5.3.1. Declaring an aspect

Using the schema support, an aspect is simply a regular Java object defined as a bean in your Spring application context. The state and behavior is captured in the fields and methods of the object, and the pointcut and advice information is captured in the XML.

An aspect is declared using the <aop:aspect> element, and the backing bean is referenced using the ref attribute:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

...

</aop:aspect>

</aop:config>

<bean id="aBean" class="...">

...

</bean>

The bean backing the aspect ("aBean" in this case) can of course be configured and dependency injected just like any other Spring bean.

5.3.2. Declaring a pointcut

A named pointcut can be declared inside an <aop:config> element, enabling the pointcut definition to be shared across several aspects and advisors.

A pointcut representing the execution of any business service in the service layer could be defined as follows:

<aop:config>

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

</aop:config>

Note that the pointcut expression itself is using the same AspectJ pointcut expression language as described in @AspectJ support. If you are using the schema based declaration style, you can refer to named pointcuts defined in types (@Aspects) within the pointcut expression. Another way of defining the above pointcut would be:

<aop:config>

<aop:pointcut id="businessService"

expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>

Assuming you have a SystemArchitecture aspect as described in Sharing common pointcut definitions.

Declaring a pointcut inside an aspect is very similar to declaring a top-level pointcut:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

...

</aop:aspect>

</aop:config>

Much the same way in an @AspectJ aspect, pointcuts declared using the schema based definition style may collect join point context. For example, the following pointcut collects the 'this' object as the join point context and passes it to advice:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..)) &amp;&amp; this(service)"/>

<aop:before pointcut-ref="businessService" method="monitor"/>

...

</aop:aspect>

</aop:config>

The advice must be declared to receive the collected join point context by including parameters of the matching names:

public void monitor(Object service) {

...

}

When combining pointcut sub-expressions, && is awkward within an XML document, and so the keywords and, or, and not can be used in place of &&, ||, and ! respectively. For example, the previous pointcut may be better written as:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service..(..)) and this(service)"/>

<aop:before pointcut-ref="businessService" method="monitor"/>

...

</aop:aspect>

</aop:config>

Note that pointcuts defined in this way are referred to by their XML id and cannot be used as named pointcuts to form composite pointcuts. The named pointcut support in the schema based definition style is thus more limited than that offered by the @AspectJ style.

5.3.3. Declaring advice

The same five advice kinds are supported as for the @AspectJ style, and they have exactly the same semantics.

Before advice

Before advice runs before a matched method execution. It is declared inside an <aop:aspect> using the <aop:before> element.

<aop:aspect id="beforeExample" ref="aBean">

<aop:before

pointcut-ref="dataAccessOperation"

method="doAccessCheck"/>

...

</aop:aspect>

Here dataAccessOperation is the id of a pointcut defined at the top ( <aop:config>) level. To define the pointcut inline instead, replace the pointcut-ref attribute with a pointcut attribute:

<aop:aspect id="beforeExample" ref="aBean">

<aop:before

pointcut="execution(\* com.xyz.myapp.dao.\*.\*(..))"

method="doAccessCheck"/>

...

</aop:aspect>

As we noted in the discussion of the @AspectJ style, using named pointcuts can significantly improve the readability of your code.

The method attribute identifies a method ( doAccessCheck) that provides the body of the advice. This method must be defined for the bean referenced by the aspect element containing the advice. Before a data access operation is executed (a method execution join point matched by the pointcut expression), the "doAccessCheck" method on the aspect bean will be invoked.

After returning advice

After returning advice runs when a matched method execution completes normally. It is declared inside an <aop:aspect> in the same way as before advice. For example:

<aop:aspect id="afterReturningExample" ref="aBean">

<aop:after-returning

pointcut-ref="dataAccessOperation"

method="doAccessCheck"/>

...

</aop:aspect>

Just as in the @AspectJ style, it is possible to get hold of the return value within the advice body. Use the returning attribute to specify the name of the parameter to which the return value should be passed:

<aop:aspect id="afterReturningExample" ref="aBean">

<aop:after-returning

pointcut-ref="dataAccessOperation"

returning="retVal"

method="doAccessCheck"/>

...

</aop:aspect>

The doAccessCheck method must declare a parameter named retVal. The type of this parameter constrains matching in the same way as described for @AfterReturning. For example, the method signature may be declared as:

public void doAccessCheck(Object retVal) {...

After throwing advice

After throwing advice executes when a matched method execution exits by throwing an exception. It is declared inside an <aop:aspect> using the after-throwing element:

<aop:aspect id="afterThrowingExample" ref="aBean">

<aop:after-throwing

pointcut-ref="dataAccessOperation"

method="doRecoveryActions"/>

...

</aop:aspect>

Just as in the @AspectJ style, it is possible to get hold of the thrown exception within the advice body. Use the throwing attribute to specify the name of the parameter to which the exception should be passed:

<aop:aspect id="afterThrowingExample" ref="aBean">

<aop:after-throwing

pointcut-ref="dataAccessOperation"

throwing="dataAccessEx"

method="doRecoveryActions"/>

...

</aop:aspect>

The doRecoveryActions method must declare a parameter named dataAccessEx. The type of this parameter constrains matching in the same way as described for @AfterThrowing. For example, the method signature may be declared as:

public void doRecoveryActions(DataAccessException dataAccessEx) {...

After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the after element:

<aop:aspect id="afterFinallyExample" ref="aBean">

<aop:after

pointcut-ref="dataAccessOperation"

method="doReleaseLock"/>

...

</aop:aspect>

Around advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements; don’t use around advice if simple before advice would do.

Around advice is declared using the aop:around element. The first parameter of the advice method must be of type ProceedingJoinPoint. Within the body of the advice, calling proceed() on the ProceedingJoinPoint causes the underlying method to execute. The proceed method may also be calling passing in an Object[] - the values in the array will be used as the arguments to the method execution when it proceeds. See Around advice for notes on calling proceed with an Object[].

<aop:aspect id="aroundExample" ref="aBean">

<aop:around

pointcut-ref="businessService"

method="doBasicProfiling"/>

...

</aop:aspect>

The implementation of the doBasicProfiling advice would be exactly the same as in the @AspectJ example (minus the annotation of course):

public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {

// start stopwatch

Object retVal = pjp.proceed();

// stop stopwatch

return retVal;

}

Advice parameters

The schema based declaration style supports fully typed advice in the same way as described for the @AspectJ support - by matching pointcut parameters by name against advice method parameters. See Advice parameters for details. If you wish to explicitly specify argument names for the advice methods (not relying on the detection strategies previously described) then this is done using the arg-names attribute of the advice element, which is treated in the same manner to the "argNames" attribute in an advice annotation as described in Determining argument names. For example:

<aop:before

pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"

method="audit"

arg-names="auditable"/>

The arg-names attribute accepts a comma-delimited list of parameter names.

Find below a slightly more involved example of the XSD-based approach that illustrates some around advice used in conjunction with a number of strongly typed parameters.

package x.y.service;

public interface FooService {

Foo getFoo(String fooName, int age);

}

public class DefaultFooService implements FooService {

public Foo getFoo(String name, int age) {

return new Foo(name, age);

}

}

Next up is the aspect. Notice the fact that the profile(..) method accepts a number of strongly-typed parameters, the first of which happens to be the join point used to proceed with the method call: the presence of this parameter is an indication that the profile(..) is to be used as around advice:

package x.y;

import org.aspectj.lang.ProceedingJoinPoint;

import org.springframework.util.StopWatch;

public class SimpleProfiler {

public Object profile(ProceedingJoinPoint call, String name, int age) throws Throwable {

StopWatch clock = new StopWatch("Profiling for '" + name + "' and '" + age + "'");

try {

clock.start(call.toShortString());

return call.proceed();

} finally {

clock.stop();

System.out.println(clock.prettyPrint());

}

}

}

Finally, here is the XML configuration that is required to effect the execution of the above advice for a particular join point:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- this is the object that will be proxied by Spring's AOP infrastructure -->

<bean id="fooService" class="x.y.service.DefaultFooService"/>

<!-- this is the actual advice itself -->

<bean id="profiler" class="x.y.SimpleProfiler"/>

<aop:config>

<aop:aspect ref="profiler">

<aop:pointcut id="theExecutionOfSomeFooServiceMethod"

expression="execution(\* x.y.service.FooService.getFoo(String,int))

and args(name, age)"/>

<aop:around pointcut-ref="theExecutionOfSomeFooServiceMethod"

method="profile"/>

</aop:aspect>

</aop:config>

</beans>

If we had the following driver script, we would get output something like this on standard output:

import org.springframework.beans.factory.BeanFactory;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import x.y.service.FooService;

public final class Boot {

public static void main(final String[] args) throws Exception {

BeanFactory ctx = new ClassPathXmlApplicationContext("x/y/plain.xml");

FooService foo = (FooService) ctx.getBean("fooService");

foo.getFoo("Pengo", 12);

}

}

StopWatch 'Profiling for 'Pengo' and '12'': running time (millis) = 0

-----------------------------------------

ms % Task name

-----------------------------------------

00000 ? execution(getFoo)

Advice ordering

When multiple advice needs to execute at the same join point (executing method) the ordering rules are as described in Advice ordering. The precedence between aspects is determined by either adding the Order annotation to the bean backing the aspect or by having the bean implement the Ordered interface.

5.3.4. Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

An introduction is made using the aop:declare-parents element inside an aop:aspect This element is used to declare that matching types have a new parent (hence the name). For example, given an interface UsageTracked, and an implementation of that interface DefaultUsageTracked, the following aspect declares that all implementors of service interfaces also implement the UsageTracked interface. (In order to expose statistics via JMX for example.)

<aop:aspect id="usageTrackerAspect" ref="usageTracking">

<aop:declare-parents

types-matching="com.xzy.myapp.service.\*+"

implement-interface="com.xyz.myapp.service.tracking.UsageTracked"

default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>

<aop:before

pointcut="com.xyz.myapp.SystemArchitecture.businessService()

and this(usageTracked)"

method="recordUsage"/>

</aop:aspect>

The class backing the usageTracking bean would contain the method:

public void recordUsage(UsageTracked usageTracked) {

usageTracked.incrementUseCount();

}

The interface to be implemented is determined by implement-interface attribute. The value of the types-matching attribute is an AspectJ type pattern :- any bean of a matching type will implement the UsageTracked interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the UsageTracked interface. If accessing a bean programmatically you would write the following:

UsageTracked usageTracked = (UsageTracked) context.getBean("myService");

5.3.5. Aspect instantiation models

The only supported instantiation model for schema-defined aspects is the singleton model. Other instantiation models may be supported in future releases.

5.3.6. Advisors

The concept of "advisors" is brought forward from the AOP support defined in Spring and does not have a direct equivalent in AspectJ. An advisor is like a small self-contained aspect that has a single piece of advice. The advice itself is represented by a bean, and must implement one of the advice interfaces described in Advice types in Spring. Advisors can take advantage of AspectJ pointcut expressions though.

Spring supports the advisor concept with the <aop:advisor> element. You will most commonly see it used in conjunction with transactional advice, which also has its own namespace support in Spring. Here’s how it looks:

<aop:config>

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

<aop:advisor

pointcut-ref="businessService"

advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">

<tx:attributes>

<tx:method name="\*" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

As well as the pointcut-ref attribute used in the above example, you can also use the pointcut attribute to define a pointcut expression inline.

To define the precedence of an advisor so that the advice can participate in ordering, use the order attribute to define the Ordered value of the advisor.

5.3.7. Example

Let’s see how the concurrent locking failure retry example from Example looks when rewritten using the schema support.

The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely it will succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don’t need to go back to the user for conflict resolution), we’d like to transparently retry the operation to avoid the client seeing a PessimisticLockingFailureException. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.

Because we want to retry the operation, we’ll need to use around advice so that we can call proceed multiple times. Here’s how the basic aspect implementation looks (it’s just a regular Java class using the schema support):

public class ConcurrentOperationExecutor implements Ordered {

private static final int DEFAULT\_MAX\_RETRIES = 2;

private int maxRetries = DEFAULT\_MAX\_RETRIES;

private int order = 1;

public void setMaxRetries(int maxRetries) {

this.maxRetries = maxRetries;

}

public int getOrder() {

return this.order;

}

public void setOrder(int order) {

this.order = order;

}

public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {

int numAttempts = 0;

PessimisticLockingFailureException lockFailureException;

do {

numAttempts++;

try {

return pjp.proceed();

}

catch(PessimisticLockingFailureException ex) {

lockFailureException = ex;

}

} while(numAttempts <= this.maxRetries);

throw lockFailureException;

}

}

Note that the aspect implements the Ordered interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The maxRetries and order properties will both be configured by Spring. The main action happens in the doConcurrentOperation around advice method. We try to proceed, and if we fail with a PessimisticLockingFailureException we simply try again unless we have exhausted all of our retry attempts.

This class is identical to the one used in the @AspectJ example, but with the annotations removed.

The corresponding Spring configuration is:

<aop:config>

<aop:aspect id="concurrentOperationRetry" ref="concurrentOperationExecutor">

<aop:pointcut id="idempotentOperation"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

<aop:around

pointcut-ref="idempotentOperation"

method="doConcurrentOperation"/>

</aop:aspect>

</aop:config>

<bean id="concurrentOperationExecutor"

class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">

<property name="maxRetries" value="3"/>

<property name="order" value="100"/>

</bean>

Notice that for the time being we assume that all business services are idempotent. If this is not the case we can refine the aspect so that it only retries genuinely idempotent operations, by introducing an Idempotent annotation:

@Retention(RetentionPolicy.RUNTIME)

public @interface Idempotent {

// marker annotation

}

and using the annotation to annotate the implementation of service operations. The change to the aspect to retry only idempotent operations simply involves refining the pointcut expression so that only @Idempotent operations match:

<aop:pointcut id="idempotentOperation"

expression="execution(\* com.xyz.myapp.service.\*.\*(..)) and

@annotation(com.xyz.myapp.service.Idempotent)"/>

5.4. Choosing which AOP declaration style to use

Once you have decided that an aspect is the best approach for implementing a given requirement, how do you decide between using Spring AOP or AspectJ, and between the Aspect language (code) style, @AspectJ annotation style, or the Spring XML style? These decisions are influenced by a number of factors including application requirements, development tools, and team familiarity with AOP.

5.4.1. Spring AOP or full AspectJ?

Use the simplest thing that can work. Spring AOP is simpler than using full AspectJ as there is no requirement to introduce the AspectJ compiler / weaver into your development and build processes. If you only need to advise the execution of operations on Spring beans, then Spring AOP is the right choice. If you need to advise objects not managed by the Spring container (such as domain objects typically), then you will need to use AspectJ. You will also need to use AspectJ if you wish to advise join points other than simple method executions (for example, field get or set join points, and so on).

When using AspectJ, you have the choice of the AspectJ language syntax (also known as the "code style") or the @AspectJ annotation style. Clearly, if you are not using Java 5+ then the choice has been made for you…​ use the code style. If aspects play a large role in your design, and you are able to use the AspectJ Development Tools (AJDT) plugin for Eclipse, then the AspectJ language syntax is the preferred option: it is cleaner and simpler because the language was purposefully designed for writing aspects. If you are not using Eclipse, or have only a few aspects that do not play a major role in your application, then you may want to consider using the @AspectJ style and sticking with a regular Java compilation in your IDE, and adding an aspect weaving phase to your build script.

5.4.2. @AspectJ or XML for Spring AOP?

If you have chosen to use Spring AOP, then you have a choice of @AspectJ or XML style. There are various tradeoffs to consider.

The XML style will be most familiar to existing Spring users and it is backed by genuine POJOs. When using AOP as a tool to configure enterprise services then XML can be a good choice (a good test is whether you consider the pointcut expression to be a part of your configuration you might want to change independently). With the XML style arguably it is clearer from your configuration what aspects are present in the system.

The XML style has two disadvantages. Firstly it does not fully encapsulate the implementation of the requirement it addresses in a single place. The DRY principle says that there should be a single, unambiguous, authoritative representation of any piece of knowledge within a system. When using the XML style, the knowledge of how a requirement is implemented is split across the declaration of the backing bean class, and the XML in the configuration file. When using the @AspectJ style there is a single module - the aspect - in which this information is encapsulated. Secondly, the XML style is slightly more limited in what it can express than the @AspectJ style: only the "singleton" aspect instantiation model is supported, and it is not possible to combine named pointcuts declared in XML. For example, in the @AspectJ style you can write something like:

@Pointcut(execution(\* get\*()))

public void propertyAccess() {}

@Pointcut(execution(org.xyz.Account+ \*(..))

public void operationReturningAnAccount() {}

@Pointcut(propertyAccess() && operationReturningAnAccount())

public void accountPropertyAccess() {}

In the XML style I can declare the first two pointcuts:

<aop:pointcut id="propertyAccess"

expression="execution(\* get\*())"/>

<aop:pointcut id="operationReturningAnAccount"

expression="execution(org.xyz.Account+ \*(..))"/>

The downside of the XML approach is that you cannot define the accountPropertyAccess pointcut by combining these definitions.

The @AspectJ style supports additional instantiation models, and richer pointcut composition. It has the advantage of keeping the aspect as a modular unit. It also has the advantage the @AspectJ aspects can be understood (and thus consumed) both by Spring AOP and by AspectJ - so if you later decide you need the capabilities of AspectJ to implement additional requirements then it is very easy to migrate to an AspectJ-based approach. On balance the Spring team prefer the @AspectJ style whenever you have aspects that do more than simple "configuration" of enterprise services.

5.5. Mixing aspect types

It is perfectly possible to mix @AspectJ style aspects using the autoproxying support, schema-defined <aop:aspect> aspects, <aop:advisor> declared advisors and even proxies and interceptors defined using the Spring 1.2 style in the same configuration. All of these are implemented using the same underlying support mechanism and will co-exist without any difficulty.

5.6. Proxying mechanisms

Spring AOP uses either JDK dynamic proxies or CGLIB to create the proxy for a given target object. (JDK dynamic proxies are preferred whenever you have a choice).

If the target object to be proxied implements at least one interface then a JDK dynamic proxy will be used. All of the interfaces implemented by the target type will be proxied. If the target object does not implement any interfaces then a CGLIB proxy will be created.

If you want to force the use of CGLIB proxying (for example, to proxy every method defined for the target object, not just those implemented by its interfaces) you can do so. However, there are some issues to consider:

final methods cannot be advised, as they cannot be overridden.

As of Spring 3.2, it is no longer necessary to add CGLIB to your project classpath, as CGLIB classes are repackaged under org.springframework and included directly in the spring-core JAR. This means that CGLIB-based proxy support 'just works' in the same way that JDK dynamic proxies always have.

As of Spring 4.0, the constructor of your proxied object will NOT be called twice anymore since the CGLIB proxy instance will be created via Objenesis. Only if your JVM does not allow for constructor bypassing, you might see double invocations and corresponding debug log entries from Spring’s AOP support.

To force the use of CGLIB proxies set the value of the proxy-target-class attribute of the <aop:config> element to true:

<aop:config proxy-target-class="true">

<!-- other beans defined here... -->

</aop:config>

To force CGLIB proxying when using the @AspectJ autoproxy support, set the 'proxy-target-class' attribute of the <aop:aspectj-autoproxy> element to true:

<aop:aspectj-autoproxy proxy-target-class="true"/>

Multiple <aop:config/> sections are collapsed into a single unified auto-proxy creator at runtime, which applies the strongest proxy settings that any of the <aop:config/> sections (typically from different XML bean definition files) specified. This also applies to the <tx:annotation-driven/> and <aop:aspectj-autoproxy/> elements.

To be clear: using proxy-target-class="true" on <tx:annotation-driven/>, <aop:aspectj-autoproxy/> or <aop:config/> elements will force the use of CGLIB proxies for all three of them.

5.6.1. Understanding AOP proxies

Spring AOP is proxy-based. It is vitally important that you grasp the semantics of what that last statement actually means before you write your own aspects or use any of the Spring AOP-based aspects supplied with the Spring Framework.

Consider first the scenario where you have a plain-vanilla, un-proxied, nothing-special-about-it, straight object reference, as illustrated by the following code snippet.

public class SimplePojo implements Pojo {

public void foo() {

// this next method invocation is a direct call on the 'this' reference

this.bar();

}

public void bar() {

// some logic...

}

}

If you invoke a method on an object reference, the method is invoked directly on that object reference, as can be seen below.

aop proxy plain pojo call

public class Main {

public static void main(String[] args) {

Pojo pojo = new SimplePojo();

// this is a direct method call on the 'pojo' reference

pojo.foo();

}

}

Things change slightly when the reference that client code has is a proxy. Consider the following diagram and code snippet.

aop proxy call

public class Main {

public static void main(String[] args) {

ProxyFactory factory = new ProxyFactory(new SimplePojo());

factory.addInterface(Pojo.class);

factory.addAdvice(new RetryAdvice());

Pojo pojo = (Pojo) factory.getProxy();

// this is a method call on the proxy!

pojo.foo();

}

}

The key thing to understand here is that the client code inside the main(..) of the Main class has a reference to the proxy. This means that method calls on that object reference will be calls on the proxy, and as such the proxy will be able to delegate to all of the interceptors (advice) that are relevant to that particular method call. However, once the call has finally reached the target object, the SimplePojo reference in this case, any method calls that it may make on itself, such as this.bar() or this.foo(), are going to be invoked against the this reference, and not the proxy. This has important implications. It means that self-invocation is not going to result in the advice associated with a method invocation getting a chance to execute.

Okay, so what is to be done about this? The best approach (the term best is used loosely here) is to refactor your code such that the self-invocation does not happen. For sure, this does entail some work on your part, but it is the best, least-invasive approach. The next approach is absolutely horrendous, and I am almost reticent to point it out precisely because it is so horrendous. You can (choke!) totally tie the logic within your class to Spring AOP by doing this:

public class SimplePojo implements Pojo {

public void foo() {

// this works, but... gah!

((Pojo) AopContext.currentProxy()).bar();

}

public void bar() {

// some logic...

}

}

This totally couples your code to Spring AOP, and it makes the class itself aware of the fact that it is being used in an AOP context, which flies in the face of AOP. It also requires some additional configuration when the proxy is being created:

public class Main {

public static void main(String[] args) {

ProxyFactory factory = new ProxyFactory(new SimplePojo());

factory.adddInterface(Pojo.class);

factory.addAdvice(new RetryAdvice());

factory.setExposeProxy(true);

Pojo pojo = (Pojo) factory.getProxy();

// this is a method call on the proxy!

pojo.foo();

}

}

Finally, it must be noted that AspectJ does not have this self-invocation issue because it is not a proxy-based AOP framework.

5.7. Programmatic creation of @AspectJ Proxies

In addition to declaring aspects in your configuration using either <aop:config> or <aop:aspectj-autoproxy>, it is also possible programmatically to create proxies that advise target objects. For the full details of Spring’s AOP API, see the next chapter. Here we want to focus on the ability to automatically create proxies using @AspectJ aspects.

The class org.springframework.aop.aspectj.annotation.AspectJProxyFactory can be used to create a proxy for a target object that is advised by one or more @AspectJ aspects. Basic usage for this class is very simple, as illustrated below. See the javadocs for full information.

// create a factory that can generate a proxy for the given target object

AspectJProxyFactory factory = new AspectJProxyFactory(targetObject);

// add an aspect, the class must be an @AspectJ aspect

// you can call this as many times as you need with different aspects

factory.addAspect(SecurityManager.class);

// you can also add existing aspect instances, the type of the object supplied must be an @AspectJ aspect

factory.addAspect(usageTracker);

// now get the proxy object...

MyInterfaceType proxy = factory.getProxy();

5.8. Using AspectJ with Spring applications

Everything we’ve covered so far in this chapter is pure Spring AOP. In this section, we’re going to look at how you can use the AspectJ compiler/weaver instead of, or in addition to, Spring AOP if your needs go beyond the facilities offered by Spring AOP alone.

Spring ships with a small AspectJ aspect library, which is available standalone in your distribution as spring-aspects.jar; you’ll need to add this to your classpath in order to use the aspects in it. Using AspectJ to dependency inject domain objects with Spring and Other Spring aspects for AspectJ discuss the content of this library and how you can use it. Configuring AspectJ aspects using Spring IoC discusses how to dependency inject AspectJ aspects that are woven using the AspectJ compiler. Finally, Load-time weaving with AspectJ in the Spring Framework provides an introduction to load-time weaving for Spring applications using AspectJ.

5.8.1. Using AspectJ to dependency inject domain objects with Spring

The Spring container instantiates and configures beans defined in your application context. It is also possible to ask a bean factory to configure a pre-existing object given the name of a bean definition containing the configuration to be applied. The spring-aspects.jar contains an annotation-driven aspect that exploits this capability to allow dependency injection of any object. The support is intended to be used for objects created outside of the control of any container. Domain objects often fall into this category because they are often created programmatically using the new operator, or by an ORM tool as a result of a database query.

The @Configurable annotation marks a class as eligible for Spring-driven configuration. In the simplest case it can be used just as a marker annotation:

package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable

public class Account {

// ...

}

When used as a marker interface in this way, Spring will configure new instances of the annotated type ( Account in this case) using a bean definition (typically prototype-scoped) with the same name as the fully-qualified type name ( com.xyz.myapp.domain.Account). Since the default name for a bean is the fully-qualified name of its type, a convenient way to declare the prototype definition is simply to omit the id attribute:

<bean class="com.xyz.myapp.domain.Account" scope="prototype">

<property name="fundsTransferService" ref="fundsTransferService"/>

</bean>

If you want to explicitly specify the name of the prototype bean definition to use, you can do so directly in the annotation:

package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable("account")

public class Account {

// ...

}

Spring will now look for a bean definition named "account" and use that as the definition to configure new Account instances.

You can also use autowiring to avoid having to specify a dedicated bean definition at all. To have Spring apply autowiring use the autowire property of the @Configurable annotation: specify either @Configurable(autowire=Autowire.BY\_TYPE) or @Configurable(autowire=Autowire.BY\_NAME for autowiring by type or by name respectively. As an alternative, as of Spring 2.5 it is preferable to specify explicit, annotation-driven dependency injection for your @Configurable beans by using @Autowired or @Inject at the field or method level (see Annotation-based container configuration for further details).

Finally you can enable Spring dependency checking for the object references in the newly created and configured object by using the dependencyCheck attribute (for example: @Configurable(autowire=Autowire.BY\_NAME,dependencyCheck=true)). If this attribute is set to true, then Spring will validate after configuration that all properties (which are not primitives or collections) have been set.

Using the annotation on its own does nothing of course. It is the AnnotationBeanConfigurerAspect in spring-aspects.jar that acts on the presence of the annotation. In essence the aspect says "after returning from the initialization of a new object of a type annotated with @Configurable, configure the newly created object using Spring in accordance with the properties of the annotation". In this context, initialization refers to newly instantiated objects (e.g., objects instantiated with the new operator) as well as to Serializable objects that are undergoing deserialization (e.g., via readResolve()).

One of the key phrases in the above paragraph is 'in essence'. For most cases, the exact semantics of 'after returning from the initialization of a new object' will be fine…​ in this context, 'after initialization' means that the dependencies will be injected after the object has been constructed - this means that the dependencies will not be available for use in the constructor bodies of the class. If you want the dependencies to be injected before the constructor bodies execute, and thus be available for use in the body of the constructors, then you need to define this on the @Configurable declaration like so:

@Configurable(preConstruction=true)

You can find out more information about the language semantics of the various pointcut types in AspectJ in this appendix of the AspectJ Programming Guide.

For this to work the annotated types must be woven with the AspectJ weaver - you can either use a build-time Ant or Maven task to do this (see for example the AspectJ Development Environment Guide) or load-time weaving (see Load-time weaving with AspectJ in the Spring Framework). The AnnotationBeanConfigurerAspect itself needs configuring by Spring (in order to obtain a reference to the bean factory that is to be used to configure new objects). If you are using Java based configuration simply add @EnableSpringConfigured to any @Configuration class.

@Configuration

@EnableSpringConfigured

public class AppConfig {

}

If you prefer XML based configuration, the Spring context namespace defines a convenient context:spring-configured element:

<context:spring-configured/>

Instances of @Configurable objects created before the aspect has been configured will result in a message being issued to the debug log and no configuration of the object taking place. An example might be a bean in the Spring configuration that creates domain objects when it is initialized by Spring. In this case you can use the "depends-on" bean attribute to manually specify that the bean depends on the configuration aspect.

<bean id="myService"

class="com.xzy.myapp.service.MyService"

depends-on="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect">

<!-- ... -->

</bean>

Do not activate @Configurable processing through the bean configurer aspect unless you really mean to rely on its semantics at runtime. In particular, make sure that you do not use @Configurable on bean classes which are registered as regular Spring beans with the container: You would get double initialization otherwise, once through the container and once through the aspect.

Unit testing @Configurable objects

One of the goals of the @Configurable support is to enable independent unit testing of domain objects without the difficulties associated with hard-coded lookups. If @Configurable types have not been woven by AspectJ then the annotation has no affect during unit testing, and you can simply set mock or stub property references in the object under test and proceed as normal. If @Configurable types have been woven by AspectJ then you can still unit test outside of the container as normal, but you will see a warning message each time that you construct an @Configurable object indicating that it has not been configured by Spring.

Working with multiple application contexts

The AnnotationBeanConfigurerAspect used to implement the @Configurable support is an AspectJ singleton aspect. The scope of a singleton aspect is the same as the scope of static members, that is to say there is one aspect instance per classloader that defines the type. This means that if you define multiple application contexts within the same classloader hierarchy you need to consider where to define the @EnableSpringConfigured bean and where to place spring-aspects.jar on the classpath.

Consider a typical Spring web-app configuration with a shared parent application context defining common business services and everything needed to support them, and one child application context per servlet containing definitions particular to that servlet. All of these contexts will co-exist within the same classloader hierarchy, and so the AnnotationBeanConfigurerAspect can only hold a reference to one of them. In this case we recommend defining the @EnableSpringConfigured bean in the shared (parent) application context: this defines the services that you are likely to want to inject into domain objects. A consequence is that you cannot configure domain objects with references to beans defined in the child (servlet-specific) contexts using the @Configurable mechanism (probably not something you want to do anyway!).

When deploying multiple web-apps within the same container, ensure that each web-application loads the types in spring-aspects.jar using its own classloader (for example, by placing spring-aspects.jar in 'WEB-INF/lib'). If spring-aspects.jar is only added to the container wide classpath (and hence loaded by the shared parent classloader), all web applications will share the same aspect instance which is probably not what you want.

5.8.2. Other Spring aspects for AspectJ

In addition to the @Configurable aspect, spring-aspects.jar contains an AspectJ aspect that can be used to drive Spring’s transaction management for types and methods annotated with the @Transactional annotation. This is primarily intended for users who want to use the Spring Framework’s transaction support outside of the Spring container.

The aspect that interprets @Transactional annotations is the AnnotationTransactionAspect. When using this aspect, you must annotate the implementation class (and/or methods within that class), not the interface (if any) that the class implements. AspectJ follows Java’s rule that annotations on interfaces are not inherited.

A @Transactional annotation on a class specifies the default transaction semantics for the execution of any public operation in the class.

A @Transactional annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). Methods of any visibility may be annotated, including private methods. Annotating non-public methods directly is the only way to get transaction demarcation for the execution of such methods.

Since Spring Framework 4.2, spring-aspects provides a similar aspect that offers the exact same features for the standard javax.transaction.Transactional annotation. Check JtaAnnotationTransactionAspect for more details.

For AspectJ programmers that want to use the Spring configuration and transaction management support but don’t want to (or cannot) use annotations, spring-aspects.jar also contains abstract aspects you can extend to provide your own pointcut definitions. See the sources for the AbstractBeanConfigurerAspect and AbstractTransactionAspect aspects for more information. As an example, the following excerpt shows how you could write an aspect to configure all instances of objects defined in the domain model using prototype bean definitions that match the fully-qualified class names:

public aspect DomainObjectConfiguration extends AbstractBeanConfigurerAspect {

public DomainObjectConfiguration() {

setBeanWiringInfoResolver(new ClassNameBeanWiringInfoResolver());

}

// the creation of a new bean (any object in the domain model)

protected pointcut beanCreation(Object beanInstance) :

initialization(new(..)) &&

SystemArchitecture.inDomainModel() &&

this(beanInstance);

}

5.8.3. Configuring AspectJ aspects using Spring IoC

When using AspectJ aspects with Spring applications, it is natural to both want and expect to be able to configure such aspects using Spring. The AspectJ runtime itself is responsible for aspect creation, and the means of configuring the AspectJ created aspects via Spring depends on the AspectJ instantiation model (the per-xxx clause) used by the aspect.

The majority of AspectJ aspects are singleton aspects. Configuration of these aspects is very easy: simply create a bean definition referencing the aspect type as normal, and include the bean attribute 'factory-method="aspectOf"'. This ensures that Spring obtains the aspect instance by asking AspectJ for it rather than trying to create an instance itself. For example:

<bean id="profiler" class="com.xyz.profiler.Profiler"

factory-method="aspectOf">

<property name="profilingStrategy" ref="jamonProfilingStrategy"/>

</bean>

Non-singleton aspects are harder to configure: however it is possible to do so by creating prototype bean definitions and using the @Configurable support from spring-aspects.jar to configure the aspect instances once they have bean created by the AspectJ runtime.

If you have some @AspectJ aspects that you want to weave with AspectJ (for example, using load-time weaving for domain model types) and other @AspectJ aspects that you want to use with Spring AOP, and these aspects are all configured using Spring, then you will need to tell the Spring AOP @AspectJ autoproxying support which exact subset of the @AspectJ aspects defined in the configuration should be used for autoproxying. You can do this by using one or more <include/> elements inside the <aop:aspectj-autoproxy/> declaration. Each <include/> element specifies a name pattern, and only beans with names matched by at least one of the patterns will be used for Spring AOP autoproxy configuration:

<aop:aspectj-autoproxy>

<aop:include name="thisBean"/>

<aop:include name="thatBean"/>

</aop:aspectj-autoproxy>

Do not be misled by the name of the <aop:aspectj-autoproxy/> element: using it will result in the creation of Spring AOP proxies. The @AspectJ style of aspect declaration is just being used here, but the AspectJ runtime is not involved.

5.8.4. Load-time weaving with AspectJ in the Spring Framework

Load-time weaving (LTW) refers to the process of weaving AspectJ aspects into an application’s class files as they are being loaded into the Java virtual machine (JVM). The focus of this section is on configuring and using LTW in the specific context of the Spring Framework: this section is not an introduction to LTW though. For full details on the specifics of LTW and configuring LTW with just AspectJ (with Spring not being involved at all), see the LTW section of the AspectJ Development Environment Guide.

The value-add that the Spring Framework brings to AspectJ LTW is in enabling much finer-grained control over the weaving process. 'Vanilla' AspectJ LTW is effected using a Java (5+) agent, which is switched on by specifying a VM argument when starting up a JVM. It is thus a JVM-wide setting, which may be fine in some situations, but often is a little too coarse. Spring-enabled LTW enables you to switch on LTW on a per-ClassLoader basis, which obviously is more fine-grained and which can make more sense in a 'single-JVM-multiple-application' environment (such as is found in a typical application server environment).

Further, in certain environments, this support enables load-time weaving without making any modifications to the application server’s launch script that will be needed to add -javaagent:path/to/aspectjweaver.jar or (as we describe later in this section) -javaagent:path/to/org.springframework.instrument-{version}.jar (previously named spring-agent.jar). Developers simply modify one or more files that form the application context to enable load-time weaving instead of relying on administrators who typically are in charge of the deployment configuration such as the launch script.

Now that the sales pitch is over, let us first walk through a quick example of AspectJ LTW using Spring, followed by detailed specifics about elements introduced in the following example. For a complete example, please see the Petclinic sample application.

A first example

Let us assume that you are an application developer who has been tasked with diagnosing the cause of some performance problems in a system. Rather than break out a profiling tool, what we are going to do is switch on a simple profiling aspect that will enable us to very quickly get some performance metrics, so that we can then apply a finer-grained profiling tool to that specific area immediately afterwards.

The example presented here uses XML style configuration, it is also possible to configure and use @AspectJ with Java Configuration. Specifically the @EnableLoadTimeWeaving annotation can be used as an alternative to <context:load-time-weaver/> (see below for details).

Here is the profiling aspect. Nothing too fancy, just a quick-and-dirty time-based profiler, using the @AspectJ-style of aspect declaration.

package foo;

import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Aspect;

import org.aspectj.lang.annotation.Around;

import org.aspectj.lang.annotation.Pointcut;

import org.springframework.util.StopWatch;

import org.springframework.core.annotation.Order;

@Aspect

public class ProfilingAspect {

@Around("methodsToBeProfiled()")

public Object profile(ProceedingJoinPoint pjp) throws Throwable {

StopWatch sw = new StopWatch(getClass().getSimpleName());

try {

sw.start(pjp.getSignature().getName());

return pjp.proceed();

} finally {

sw.stop();

System.out.println(sw.prettyPrint());

}

}

@Pointcut("execution(public \* foo..\*.\*(..))")

public void methodsToBeProfiled(){}

}

We will also need to create an META-INF/aop.xml file, to inform the AspectJ weaver that we want to weave our ProfilingAspect into our classes. This file convention, namely the presence of a file (or files) on the Java classpath called META-INF/aop.xml is standard AspectJ.

<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>

<weaver>

<!-- only weave classes in our application-specific packages -->

<include within="foo.\*"/>

</weaver>

<aspects>

<!-- weave in just this aspect -->

<aspect name="foo.ProfilingAspect"/>

</aspects>

</aspectj>

Now to the Spring-specific portion of the configuration. We need to configure a LoadTimeWeaver (all explained later, just take it on trust for now). This load-time weaver is the essential component responsible for weaving the aspect configuration in one or more META-INF/aop.xml files into the classes in your application. The good thing is that it does not require a lot of configuration, as can be seen below (there are some more options that you can specify, but these are detailed later).

<?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">

<!-- a service object; we will be profiling its methods -->

<bean id="entitlementCalculationService"

class="foo.StubEntitlementCalculationService"/>

<!-- this switches on the load-time weaving -->

<context:load-time-weaver/>

</beans>

Now that all the required artifacts are in place - the aspect, the META-INF/aop.xml file, and the Spring configuration -, let us create a simple driver class with a main(..) method to demonstrate the LTW in action.

package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

public static void main(String[] args) {

ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml", Main.class);

EntitlementCalculationService entitlementCalculationService

= (EntitlementCalculationService) ctx.getBean("entitlementCalculationService");

// the profiling aspect is 'woven' around this method execution

entitlementCalculationService.calculateEntitlement();

}

}

There is one last thing to do. The introduction to this section did say that one could switch on LTW selectively on a per- ClassLoader basis with Spring, and this is true. However, just for this example, we are going to use a Java agent (supplied with Spring) to switch on the LTW. This is the command line we will use to run the above Main class:

java -javaagent:C:/projects/foo/lib/global/spring-instrument.jar foo.Main

The -javaagent is a flag for specifying and enabling agents to instrument programs running on the JVM. The Spring Framework ships with such an agent, the InstrumentationSavingAgent, which is packaged in the spring-instrument.jar that was supplied as the value of the -javaagent argument in the above example.

The output from the execution of the Main program will look something like that below. (I have introduced a Thread.sleep(..) statement into the calculateEntitlement() implementation so that the profiler actually captures something other than 0 milliseconds - the 01234 milliseconds is not an overhead introduced by the AOP :) )

Calculating entitlement

StopWatch 'ProfilingAspect': running time (millis) = 1234

------ ----- ----------------------------

ms % Task name

------ ----- ----------------------------

01234 100% calculateEntitlement

Since this LTW is effected using full-blown AspectJ, we are not just limited to advising Spring beans; the following slight variation on the Main program will yield the same result.

package foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Main {

public static void main(String[] args) {

new ClassPathXmlApplicationContext("beans.xml", Main.class);

EntitlementCalculationService entitlementCalculationService =

new StubEntitlementCalculationService();

// the profiling aspect will be 'woven' around this method execution

entitlementCalculationService.calculateEntitlement();

}

}

Notice how in the above program we are simply bootstrapping the Spring container, and then creating a new instance of the StubEntitlementCalculationService totally outside the context of Spring…​ the profiling advice still gets woven in.

The example admittedly is simplistic…​ however the basics of the LTW support in Spring have all been introduced in the above example, and the rest of this section will explain the 'why' behind each bit of configuration and usage in detail.

The ProfilingAspect used in this example may be basic, but it is quite useful. It is a nice example of a development-time aspect that developers can use during development (of course), and then quite easily exclude from builds of the application being deployed into UAT or production.

Aspects

The aspects that you use in LTW have to be AspectJ aspects. They can be written in either the AspectJ language itself or you can write your aspects in the @AspectJ-style. It means that your aspects are then both valid AspectJ and Spring AOP aspects. Furthermore, the compiled aspect classes need to be available on the classpath.

'META-INF/aop.xml'

The AspectJ LTW infrastructure is configured using one or more META-INF/aop.xml files, that are on the Java classpath (either directly, or more typically in jar files).

The structure and contents of this file is detailed in the main AspectJ reference documentation, and the interested reader is referred to that resource. (I appreciate that this section is brief, but the aop.xml file is 100% AspectJ - there is no Spring-specific information or semantics that apply to it, and so there is no extra value that I can contribute either as a result), so rather than rehash the quite satisfactory section that the AspectJ developers wrote, I am just directing you there.)

Required libraries (JARS)

At a minimum you will need the following libraries to use the Spring Framework’s support for AspectJ LTW:

spring-aop.jar (version 2.5 or later, plus all mandatory dependencies)

aspectjweaver.jar (version 1.6.8 or later)

If you are using the Spring-provided agent to enable instrumentation, you will also need:

spring-instrument.jar

Spring configuration

The key component in Spring’s LTW support is the LoadTimeWeaver interface (in the org.springframework.instrument.classloading package), and the numerous implementations of it that ship with the Spring distribution. A LoadTimeWeaver is responsible for adding one or more java.lang.instrument.ClassFileTransformers to a ClassLoader at runtime, which opens the door to all manner of interesting applications, one of which happens to be the LTW of aspects.

If you are unfamiliar with the idea of runtime class file transformation, you are encouraged to read the javadoc API documentation for the java.lang.instrument package before continuing. This is not a huge chore because there is - rather annoyingly - precious little documentation there…​ the key interfaces and classes will at least be laid out in front of you for reference as you read through this section.

Configuring a LoadTimeWeaver for a particular ApplicationContext can be as easy as adding one line. (Please note that you almost certainly will need to be using an ApplicationContext as your Spring container - typically a BeanFactory will not be enough because the LTW support makes use of BeanFactoryPostProcessors.)

To enable the Spring Framework’s LTW support, you need to configure a LoadTimeWeaver, which typically is done using the @EnableLoadTimeWeaving annotation.

@Configuration

@EnableLoadTimeWeaving

public class AppConfig {

}

Alternatively, if you prefer XML based configuration, use the <context:load-time-weaver/> element. Note that the element is defined in the context namespace.

<?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:load-time-weaver/>

</beans>

The above configuration will define and register a number of LTW-specific infrastructure beans for you automatically, such as a LoadTimeWeaver and an AspectJWeavingEnabler. The default LoadTimeWeaver is the DefaultContextLoadTimeWeaver class, which attempts to decorate an automatically detected LoadTimeWeaver: the exact type of LoadTimeWeaver that will be 'automatically detected' is dependent upon your runtime environment (summarized in the following table).

Table 13. DefaultContextLoadTimeWeaver LoadTimeWeavers

Runtime Environment LoadTimeWeaver implementation

Running in Oracle’s WebLogic

WebLogicLoadTimeWeaver

Running in Oracle’s GlassFish

GlassFishLoadTimeWeaver

Running in Apache Tomcat

TomcatLoadTimeWeaver

Running in Red Hat’s JBoss AS or WildFly

JBossLoadTimeWeaver

Running in IBM’s WebSphere

WebSphereLoadTimeWeaver

JVM started with Spring InstrumentationSavingAgent (java -javaagent:path/to/spring-instrument.jar)

InstrumentationLoadTimeWeaver

Fallback, expecting the underlying ClassLoader to follow common conventions (e.g. applicable to TomcatInstrumentableClassLoader and Resin)

ReflectiveLoadTimeWeaver

Note that these are just the LoadTimeWeavers that are autodetected when using the DefaultContextLoadTimeWeaver: it is of course possible to specify exactly which LoadTimeWeaver implementation that you wish to use.

To specify a specific LoadTimeWeaver with Java configuration implement the LoadTimeWeavingConfigurer interface and override the getLoadTimeWeaver() method:

@Configuration

@EnableLoadTimeWeaving

public class AppConfig implements LoadTimeWeavingConfigurer {

@Override

public LoadTimeWeaver getLoadTimeWeaver() {

return new ReflectiveLoadTimeWeaver();

}

}

If you are using XML based configuration you can specify the fully-qualified classname as the value of the weaver-class attribute on the <context:load-time-weaver/> element:

<?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:load-time-weaver

weaver-class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>

</beans>

The LoadTimeWeaver that is defined and registered by the configuration can be later retrieved from the Spring container using the well-known name loadTimeWeaver. Remember that the LoadTimeWeaver exists just as a mechanism for Spring’s LTW infrastructure to add one or more ClassFileTransformers. The actual ClassFileTransformer that does the LTW is the ClassPreProcessorAgentAdapter (from the org.aspectj.weaver.loadtime package) class. See the class-level javadocs of the ClassPreProcessorAgentAdapter class for further details, because the specifics of how the weaving is actually effected is beyond the scope of this section.

There is one final attribute of the configuration left to discuss: the aspectjWeaving attribute (or aspectj-weaving if you are using XML). This is a simple attribute that controls whether LTW is enabled or not; it is as simple as that. It accepts one of three possible values, summarized below, with the default value being autodetect if the attribute is not present.

Table 14. AspectJ weaving attribute values

Annotation Value XML Value Explanation

ENABLED

on

AspectJ weaving is on, and aspects will be woven at load-time as appropriate.

DISABLED

off

LTW is off…​ no aspect will be woven at load-time.

AUTODETECT

autodetect

If the Spring LTW infrastructure can find at least one META-INF/aop.xml file, then AspectJ weaving is on, else it is off. This is the default value.

Environment-specific configuration

This last section contains any additional settings and configuration that you will need when using Spring’s LTW support in environments such as application servers and web containers.

Tomcat

Historically, Apache Tomcat's default class loader did not support class transformation which is why Spring provides an enhanced implementation that addresses this need. Named TomcatInstrumentableClassLoader, the loader works on Tomcat 6.0 and above.

Do not define TomcatInstrumentableClassLoader anymore on Tomcat 8.0 and higher. Instead, let Spring automatically use Tomcat’s new native InstrumentableClassLoader facility through the TomcatLoadTimeWeaver strategy.

If you still need to use TomcatInstrumentableClassLoader, it can be registered individually for each web application as follows:

Copy org.springframework.instrument.tomcat.jar into $CATALINA\_HOME/lib, where $CATALINA\_HOME represents the root of the Tomcat installation)

Instruct Tomcat to use the custom class loader (instead of the default) by editing the web application context file:

<Context path="/myWebApp" docBase="/my/webApp/location">

<Loader

loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"/>

</Context>

Apache Tomcat (6.0+) supports several context locations:

server configuration file - $CATALINA\_HOME/conf/server.xml

default context configuration - $CATALINA\_HOME/conf/context.xml - that affects all deployed web applications

per-web application configuration which can be deployed either on the server-side at $CATALINA\_HOME/conf/[enginename]/[hostname]/[webapp]-context.xml or embedded inside the web-app archive at META-INF/context.xml

For efficiency, the embedded per-web-app configuration style is recommended because it will impact only applications that use the custom class loader and does not require any changes to the server configuration. See the Tomcat 6.0.x documentation for more details about available context locations.

Alternatively, consider the use of the Spring-provided generic VM agent, to be specified in Tomcat’s launch script (see above). This will make instrumentation available to all deployed web applications, no matter what ClassLoader they happen to run on.

WebLogic, WebSphere, Resin, GlassFish, JBoss

Recent versions of WebLogic Server (version 10 and above), IBM WebSphere Application Server (version 7 and above), Resin (3.1 and above) and JBoss (6.x or above) provide a ClassLoader that is capable of local instrumentation. Spring’s native LTW leverages such ClassLoaders to enable AspectJ weaving. You can enable LTW by simply activating load-time weaving as described earlier. Specifically, you do not need to modify the launch script to add -javaagent:path/to/spring-instrument.jar.

Note that GlassFish instrumentation-capable ClassLoader is available only in its EAR environment. For GlassFish web applications, follow the Tomcat setup instructions as outlined above.

Note that on JBoss 6.x, the app server scanning needs to be disabled to prevent it from loading the classes before the application actually starts. A quick workaround is to add to your artifact a file named WEB-INF/jboss-scanning.xml with the following content:

<scanning xmlns="urn:jboss:scanning:1.0"/>

Generic Java applications

When class instrumentation is required in environments that do not support or are not supported by the existing LoadTimeWeaver implementations, a JDK agent can be the only solution. For such cases, Spring provides InstrumentationLoadTimeWeaver, which requires a Spring-specific (but very general) VM agent, org.springframework.instrument-{version}.jar (previously named spring-agent.jar).

To use it, you must start the virtual machine with the Spring agent, by supplying the following JVM options:

-javaagent:/path/to/org.springframework.instrument-{version}.jar

Note that this requires modification of the VM launch script which may prevent you from using this in application server environments (depending on your operation policies). Additionally, the JDK agent will instrument the entire VM which can prove expensive.

For performance reasons, it is recommended to use this configuration only if your target environment (such as Jetty) does not have (or does not support) a dedicated LTW.

5.9. Further Resources

More information on AspectJ can be found on the AspectJ website.

The book Eclipse AspectJ by Adrian Colyer et. al. (Addison-Wesley, 2005) provides a comprehensive introduction and reference for the AspectJ language.

The book AspectJ in Action, Second Edition by Ramnivas Laddad (Manning, 2009) comes highly recommended; the focus of the book is on AspectJ, but a lot of general AOP themes are explored (in some depth).

6. Spring AOP APIs

6.1. Introduction

The previous chapter described the Spring’s support for AOP using @AspectJ and schema-based aspect definitions. In this chapter we discuss the lower-level Spring AOP APIs and the AOP support typically used in Spring 1.2 applications. For new applications, we recommend the use of the Spring 2.0 and later AOP support described in the previous chapter, but when working with existing applications, or when reading books and articles, you may come across Spring 1.2 style examples. Spring 5 remains backwards compatible with Spring 1.2 and everything described in this chapter is fully supported in Spring 5.

6.2. Pointcut API in Spring

Let’s look at how Spring handles the crucial pointcut concept.

6.2.1. Concepts

Spring’s pointcut model enables pointcut reuse independent of advice types. It’s possible to target different advice using the same pointcut.

The org.springframework.aop.Pointcut interface is the central interface, used to target advices to particular classes and methods. The complete interface is shown below:

public interface Pointcut {

ClassFilter getClassFilter();

MethodMatcher getMethodMatcher();

}

Splitting the Pointcut interface into two parts allows reuse of class and method matching parts, and fine-grained composition operations (such as performing a "union" with another method matcher).

The ClassFilter interface is used to restrict the pointcut to a given set of target classes. If the matches() method always returns true, all target classes will be matched:

public interface ClassFilter {

boolean matches(Class clazz);

}

The MethodMatcher interface is normally more important. The complete interface is shown below:

public interface MethodMatcher {

boolean matches(Method m, Class targetClass);

boolean isRuntime();

boolean matches(Method m, Class targetClass, Object[] args);

}

The matches(Method, Class) method is used to test whether this pointcut will ever match a given method on a target class. This evaluation can be performed when an AOP proxy is created, to avoid the need for a test on every method invocation. If the 2-argument matches method returns true for a given method, and the isRuntime() method for the MethodMatcher returns true, the 3-argument matches method will be invoked on every method invocation. This enables a pointcut to look at the arguments passed to the method invocation immediately before the target advice is to execute.

Most MethodMatchers are static, meaning that their isRuntime() method returns false. In this case, the 3-argument matches method will never be invoked.

If possible, try to make pointcuts static, allowing the AOP framework to cache the results of pointcut evaluation when an AOP proxy is created.

6.2.2. Operations on pointcuts

Spring supports operations on pointcuts: notably, union and intersection.

Union means the methods that either pointcut matches.

Intersection means the methods that both pointcuts match.

Union is usually more useful.

Pointcuts can be composed using the static methods in the org.springframework.aop.support.Pointcuts class, or using the ComposablePointcut class in the same package. However, using AspectJ pointcut expressions is usually a simpler approach.

6.2.3. AspectJ expression pointcuts

Since 2.0, the most important type of pointcut used by Spring is org.springframework.aop.aspectj.AspectJExpressionPointcut. This is a pointcut that uses an AspectJ supplied library to parse an AspectJ pointcut expression string.

See the previous chapter for a discussion of supported AspectJ pointcut primitives.

6.2.4. Convenience pointcut implementations

Spring provides several convenient pointcut implementations. Some can be used out of the box; others are intended to be subclassed in application-specific pointcuts.

Static pointcuts

Static pointcuts are based on method and target class, and cannot take into account the method’s arguments. Static pointcuts are sufficient - and best - for most usages. It’s possible for Spring to evaluate a static pointcut only once, when a method is first invoked: after that, there is no need to evaluate the pointcut again with each method invocation.

Let’s consider some static pointcut implementations included with Spring.

Regular expression pointcuts

One obvious way to specify static pointcuts is regular expressions. Several AOP frameworks besides Spring make this possible. org.springframework.aop.support.JdkRegexpMethodPointcut is a generic regular expression pointcut, using the regular expression support in the JDK.

Using the JdkRegexpMethodPointcut class, you can provide a list of pattern Strings. If any of these is a match, the pointcut will evaluate to true. (So the result is effectively the union of these pointcuts.)

The usage is shown below:

<bean id="settersAndAbsquatulatePointcut"

class="org.springframework.aop.support.JdkRegexpMethodPointcut">

<property name="patterns">

<list>

<value>.\*set.\*</value>

<value>.\*absquatulate</value>

</list>

</property>

</bean>

Spring provides a convenience class, RegexpMethodPointcutAdvisor, that allows us to also reference an Advice (remember that an Advice can be an interceptor, before advice, throws advice etc.). Behind the scenes, Spring will use a JdkRegexpMethodPointcut. Using RegexpMethodPointcutAdvisor simplifies wiring, as the one bean encapsulates both pointcut and advice, as shown below:

<bean id="settersAndAbsquatulateAdvisor"

class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

<property name="advice">

<ref bean="beanNameOfAopAllianceInterceptor"/>

</property>

<property name="patterns">

<list>

<value>.\*set.\*</value>

<value>.\*absquatulate</value>

</list>

</property>

</bean>

RegexpMethodPointcutAdvisor can be used with any Advice type.

Attribute-driven pointcuts

An important type of static pointcut is a metadata-driven pointcut. This uses the values of metadata attributes: typically, source-level metadata.

Dynamic pointcuts

Dynamic pointcuts are costlier to evaluate than static pointcuts. They take into account method arguments, as well as static information. This means that they must be evaluated with every method invocation; the result cannot be cached, as arguments will vary.

The main example is the control flow pointcut.

Control flow pointcuts

Spring control flow pointcuts are conceptually similar to AspectJ cflow pointcuts, although less powerful. (There is currently no way to specify that a pointcut executes below a join point matched by another pointcut.) A control flow pointcut matches the current call stack. For example, it might fire if the join point was invoked by a method in the com.mycompany.web package, or by the SomeCaller class. Control flow pointcuts are specified using the org.springframework.aop.support.ControlFlowPointcut class.

Control flow pointcuts are significantly more expensive to evaluate at runtime than even other dynamic pointcuts. In Java 1.4, the cost is about 5 times that of other dynamic pointcuts.

6.2.5. Pointcut superclasses

Spring provides useful pointcut superclasses to help you to implement your own pointcuts.

Because static pointcuts are most useful, you’ll probably subclass StaticMethodMatcherPointcut, as shown below. This requires implementing just one abstract method (although it’s possible to override other methods to customize behavior):

class TestStaticPointcut extends StaticMethodMatcherPointcut {

public boolean matches(Method m, Class targetClass) {

// return true if custom criteria match

}

}

There are also superclasses for dynamic pointcuts.

You can use custom pointcuts with any advice type in Spring 1.0 RC2 and above.

6.2.6. Custom pointcuts

Because pointcuts in Spring AOP are Java classes, rather than language features (as in AspectJ) it’s possible to declare custom pointcuts, whether static or dynamic. Custom pointcuts in Spring can be arbitrarily complex. However, using the AspectJ pointcut expression language is recommended if possible.

Later versions of Spring may offer support for "semantic pointcuts" as offered by JAC: for example, "all methods that change instance variables in the target object."

6.3. Advice API in Spring

Let’s now look at how Spring AOP handles advice.

6.3.1. Advice lifecycles

Each advice is a Spring bean. An advice instance can be shared across all advised objects, or unique to each advised object. This corresponds to per-class or per-instance advice.

Per-class advice is used most often. It is appropriate for generic advice such as transaction advisors. These do not depend on the state of the proxied object or add new state; they merely act on the method and arguments.

Per-instance advice is appropriate for introductions, to support mixins. In this case, the advice adds state to the proxied object.

It’s possible to use a mix of shared and per-instance advice in the same AOP proxy.

6.3.2. Advice types in Spring

Spring provides several advice types out of the box, and is extensible to support arbitrary advice types. Let us look at the basic concepts and standard advice types.

Interception around advice

The most fundamental advice type in Spring is interception around advice.

Spring is compliant with the AOP Alliance interface for around advice using method interception. MethodInterceptors implementing around advice should implement the following interface:

public interface MethodInterceptor extends Interceptor {

Object invoke(MethodInvocation invocation) throws Throwable;

}

The MethodInvocation argument to the invoke() method exposes the method being invoked; the target join point; the AOP proxy; and the arguments to the method. The invoke() method should return the invocation’s result: the return value of the join point.

A simple MethodInterceptor implementation looks as follows:

public class DebugInterceptor implements MethodInterceptor {

public Object invoke(MethodInvocation invocation) throws Throwable {

System.out.println("Before: invocation=[" + invocation + "]");

Object rval = invocation.proceed();

System.out.println("Invocation returned");

return rval;

}

}

Note the call to the MethodInvocation’s proceed() method. This proceeds down the interceptor chain towards the join point. Most interceptors will invoke this method, and return its return value. However, a MethodInterceptor, like any around advice, can return a different value or throw an exception rather than invoke the proceed method. However, you don’t want to do this without good reason!

MethodInterceptors offer interoperability with other AOP Alliance-compliant AOP implementations. The other advice types discussed in the remainder of this section implement common AOP concepts, but in a Spring-specific way. While there is an advantage in using the most specific advice type, stick with MethodInterceptor around advice if you are likely to want to run the aspect in another AOP framework. Note that pointcuts are not currently interoperable between frameworks, and the AOP Alliance does not currently define pointcut interfaces.

Before advice

A simpler advice type is a before advice. This does not need a MethodInvocation object, since it will only be called before entering the method.

The main advantage of a before advice is that there is no need to invoke the proceed() method, and therefore no possibility of inadvertently failing to proceed down the interceptor chain.

The MethodBeforeAdvice interface is shown below. (Spring’s API design would allow for field before advice, although the usual objects apply to field interception and it’s unlikely that Spring will ever implement it).

public interface MethodBeforeAdvice extends BeforeAdvice {

void before(Method m, Object[] args, Object target) throws Throwable;

}

Note the return type is void. Before advice can insert custom behavior before the join point executes, but cannot change the return value. If a before advice throws an exception, this will abort further execution of the interceptor chain. The exception will propagate back up the interceptor chain. If it is unchecked, or on the signature of the invoked method, it will be passed directly to the client; otherwise it will be wrapped in an unchecked exception by the AOP proxy.

An example of a before advice in Spring, which counts all method invocations:

public class CountingBeforeAdvice implements MethodBeforeAdvice {

private int count;

public void before(Method m, Object[] args, Object target) throws Throwable {

++count;

}

public int getCount() {

return count;

}

}

Before advice can be used with any pointcut.

Throws advice

Throws advice is invoked after the return of the join point if the join point threw an exception. Spring offers typed throws advice. Note that this means that the org.springframework.aop.ThrowsAdvice interface does not contain any methods: It is a tag interface identifying that the given object implements one or more typed throws advice methods. These should be in the form of:

afterThrowing([Method, args, target], subclassOfThrowable)

Only the last argument is required. The method signatures may have either one or four arguments, depending on whether the advice method is interested in the method and arguments. The following classes are examples of throws advice.

The advice below is invoked if a RemoteException is thrown (including subclasses):

public class RemoteThrowsAdvice implements ThrowsAdvice {

public void afterThrowing(RemoteException ex) throws Throwable {

// Do something with remote exception

}

}

The following advice is invoked if a ServletException is thrown. Unlike the above advice, it declares 4 arguments, so that it has access to the invoked method, method arguments and target object:

public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {

// Do something with all arguments

}

}

The final example illustrates how these two methods could be used in a single class, which handles both RemoteException and ServletException. Any number of throws advice methods can be combined in a single class.

public static class CombinedThrowsAdvice implements ThrowsAdvice {

public void afterThrowing(RemoteException ex) throws Throwable {

// Do something with remote exception

}

public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {

// Do something with all arguments

}

}

If a throws-advice method throws an exception itself, it will override the original exception (i.e. change the exception thrown to the user). The overriding exception will typically be a RuntimeException; this is compatible with any method signature. However, if a throws-advice method throws a checked exception, it will have to match the declared exceptions of the target method and is hence to some degree coupled to specific target method signatures. Do not throw an undeclared checked exception that is incompatible with the target method’s signature!

Throws advice can be used with any pointcut.

After Returning advice

An after returning advice in Spring must implement the org.springframework.aop.AfterReturningAdvice interface, shown below:

public interface AfterReturningAdvice extends Advice {

void afterReturning(Object returnValue, Method m, Object[] args, Object target)

throws Throwable;

}

An after returning advice has access to the return value (which it cannot modify), invoked method, methods arguments and target.

The following after returning advice counts all successful method invocations that have not thrown exceptions:

public class CountingAfterReturningAdvice implements AfterReturningAdvice {

private int count;

public void afterReturning(Object returnValue, Method m, Object[] args, Object target)

throws Throwable {

++count;

}

public int getCount() {

return count;

}

}

This advice doesn’t change the execution path. If it throws an exception, this will be thrown up the interceptor chain instead of the return value.

After returning advice can be used with any pointcut.

Introduction advice

Spring treats introduction advice as a special kind of interception advice.

Introduction requires an IntroductionAdvisor, and an IntroductionInterceptor, implementing the following interface:

public interface IntroductionInterceptor extends MethodInterceptor {

boolean implementsInterface(Class intf);

}

The invoke() method inherited from the AOP Alliance MethodInterceptor interface must implement the introduction: that is, if the invoked method is on an introduced interface, the introduction interceptor is responsible for handling the method call - it cannot invoke proceed().

Introduction advice cannot be used with any pointcut, as it applies only at class, rather than method, level. You can only use introduction advice with the IntroductionAdvisor, which has the following methods:

public interface IntroductionAdvisor extends Advisor, IntroductionInfo {

ClassFilter getClassFilter();

void validateInterfaces() throws IllegalArgumentException;

}

public interface IntroductionInfo {

Class[] getInterfaces();

}

There is no MethodMatcher, and hence no Pointcut, associated with introduction advice. Only class filtering is logical.

The getInterfaces() method returns the interfaces introduced by this advisor.

The validateInterfaces() method is used internally to see whether or not the introduced interfaces can be implemented by the configured IntroductionInterceptor.

Let’s look at a simple example from the Spring test suite. Let’s suppose we want to introduce the following interface to one or more objects:

public interface Lockable {

void lock();

void unlock();

boolean locked();

}

This illustrates a mixin. We want to be able to cast advised objects to Lockable, whatever their type, and call lock and unlock methods. If we call the lock() method, we want all setter methods to throw a LockedException. Thus we can add an aspect that provides the ability to make objects immutable, without them having any knowledge of it: a good example of AOP.

Firstly, we’ll need an IntroductionInterceptor that does the heavy lifting. In this case, we extend the org.springframework.aop.support.DelegatingIntroductionInterceptor convenience class. We could implement IntroductionInterceptor directly, but using DelegatingIntroductionInterceptor is best for most cases.

The DelegatingIntroductionInterceptor is designed to delegate an introduction to an actual implementation of the introduced interface(s), concealing the use of interception to do so. The delegate can be set to any object using a constructor argument; the default delegate (when the no-arg constructor is used) is this. Thus in the example below, the delegate is the LockMixin subclass of DelegatingIntroductionInterceptor. Given a delegate (by default itself), a DelegatingIntroductionInterceptor instance looks for all interfaces implemented by the delegate (other than IntroductionInterceptor), and will support introductions against any of them. It’s possible for subclasses such as LockMixin to call the suppressInterface(Class intf) method to suppress interfaces that should not be exposed. However, no matter how many interfaces an IntroductionInterceptor is prepared to support, the IntroductionAdvisor used will control which interfaces are actually exposed. An introduced interface will conceal any implementation of the same interface by the target.

Thus LockMixin extends DelegatingIntroductionInterceptor and implements Lockable itself. The superclass automatically picks up that Lockable can be supported for introduction, so we don’t need to specify that. We could introduce any number of interfaces in this way.

Note the use of the locked instance variable. This effectively adds additional state to that held in the target object.

public class LockMixin extends DelegatingIntroductionInterceptor implements Lockable {

private boolean locked;

public void lock() {

this.locked = true;

}

public void unlock() {

this.locked = false;

}

public boolean locked() {

return this.locked;

}

public Object invoke(MethodInvocation invocation) throws Throwable {

if (locked() && invocation.getMethod().getName().indexOf("set") == 0) {

throw new LockedException();

}

return super.invoke(invocation);

}

}

Often it isn’t necessary to override the invoke() method: the DelegatingIntroductionInterceptor implementation - which calls the delegate method if the method is introduced, otherwise proceeds towards the join point - is usually sufficient. In the present case, we need to add a check: no setter method can be invoked if in locked mode.

The introduction advisor required is simple. All it needs to do is hold a distinct LockMixin instance, and specify the introduced interfaces - in this case, just Lockable. A more complex example might take a reference to the introduction interceptor (which would be defined as a prototype): in this case, there’s no configuration relevant for a LockMixin, so we simply create it using new.

public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

public LockMixinAdvisor() {

super(new LockMixin(), Lockable.class);

}

}

We can apply this advisor very simply: it requires no configuration. (However, it is necessary: It’s impossible to use an IntroductionInterceptor without an IntroductionAdvisor.) As usual with introductions, the advisor must be per-instance, as it is stateful. We need a different instance of LockMixinAdvisor, and hence LockMixin, for each advised object. The advisor comprises part of the advised object’s state.

We can apply this advisor programmatically, using the Advised.addAdvisor() method, or (the recommended way) in XML configuration, like any other advisor. All proxy creation choices discussed below, including "auto proxy creators," correctly handle introductions and stateful mixins.

6.4. Advisor API in Spring

In Spring, an Advisor is an aspect that contains just a single advice object associated with a pointcut expression.

Apart from the special case of introductions, any advisor can be used with any advice. org.springframework.aop.support.DefaultPointcutAdvisor is the most commonly used advisor class. For example, it can be used with a MethodInterceptor, BeforeAdvice or ThrowsAdvice.

It is possible to mix advisor and advice types in Spring in the same AOP proxy. For example, you could use a interception around advice, throws advice and before advice in one proxy configuration: Spring will automatically create the necessary interceptor chain.

6.5. Using the ProxyFactoryBean to create AOP proxies

If you’re using the Spring IoC container (an ApplicationContext or BeanFactory) for your business objects - and you should be! - you will want to use one of Spring’s AOP FactoryBeans. (Remember that a factory bean introduces a layer of indirection, enabling it to create objects of a different type.)

The Spring AOP support also uses factory beans under the covers.

The basic way to create an AOP proxy in Spring is to use the org.springframework.aop.framework.ProxyFactoryBean. This gives complete control over the pointcuts and advice that will apply, and their ordering. However, there are simpler options that are preferable if you don’t need such control.

6.5.1. Basics

The ProxyFactoryBean, like other Spring FactoryBean implementations, introduces a level of indirection. If you define a ProxyFactoryBean with name foo, what objects referencing foo see is not the ProxyFactoryBean instance itself, but an object created by the ProxyFactoryBean's implementation of the getObject() method. This method will create an AOP proxy wrapping a target object.

One of the most important benefits of using a ProxyFactoryBean or another IoC-aware class to create AOP proxies, is that it means that advices and pointcuts can also be managed by IoC. This is a powerful feature, enabling certain approaches that are hard to achieve with other AOP frameworks. For example, an advice may itself reference application objects (besides the target, which should be available in any AOP framework), benefiting from all the pluggability provided by Dependency Injection.

6.5.2. JavaBean properties

In common with most FactoryBean implementations provided with Spring, the ProxyFactoryBean class is itself a JavaBean. Its properties are used to:

Specify the target you want to proxy.

Specify whether to use CGLIB (see below and also JDK- and CGLIB-based proxies).

Some key properties are inherited from org.springframework.aop.framework.ProxyConfig (the superclass for all AOP proxy factories in Spring). These key properties include:

proxyTargetClass: true if the target class is to be proxied, rather than the target class' interfaces. If this property value is set to true, then CGLIB proxies will be created (but see also JDK- and CGLIB-based proxies).

optimize: controls whether or not aggressive optimizations are applied to proxies created via CGLIB. One should not blithely use this setting unless one fully understands how the relevant AOP proxy handles optimization. This is currently used only for CGLIB proxies; it has no effect with JDK dynamic proxies.

frozen: if a proxy configuration is frozen, then changes to the configuration are no longer allowed. This is useful both as a slight optimization and for those cases when you don’t want callers to be able to manipulate the proxy (via the Advised interface) after the proxy has been created. The default value of this property is false, so changes such as adding additional advice are allowed.

exposeProxy: determines whether or not the current proxy should be exposed in a ThreadLocal so that it can be accessed by the target. If a target needs to obtain the proxy and the exposeProxy property is set to true, the target can use the AopContext.currentProxy() method.

Other properties specific to ProxyFactoryBean include:

proxyInterfaces: array of String interface names. If this isn’t supplied, a CGLIB proxy for the target class will be used (but see also JDK- and CGLIB-based proxies).

interceptorNames: String array of Advisor, interceptor or other advice names to apply. Ordering is significant, on a first come-first served basis. That is to say that the first interceptor in the list will be the first to be able to intercept the invocation.

The names are bean names in the current factory, including bean names from ancestor factories. You can’t mention bean references here since doing so would result in the ProxyFactoryBean ignoring the singleton setting of the advice.

You can append an interceptor name with an asterisk ( \*). This will result in the application of all advisor beans with names starting with the part before the asterisk to be applied. An example of using this feature can be found in Using 'global' advisors.

singleton: whether or not the factory should return a single object, no matter how often the getObject() method is called. Several FactoryBean implementations offer such a method. The default value is true. If you want to use stateful advice - for example, for stateful mixins - use prototype advices along with a singleton value of false.

6.5.3. JDK- and CGLIB-based proxies

This section serves as the definitive documentation on how the ProxyFactoryBean chooses to create one of either a JDK- and CGLIB-based proxy for a particular target object (that is to be proxied).

The behavior of the ProxyFactoryBean with regard to creating JDK- or CGLIB-based proxies changed between versions 1.2.x and 2.0 of Spring. The ProxyFactoryBean now exhibits similar semantics with regard to auto-detecting interfaces as those of the TransactionProxyFactoryBean class.

If the class of a target object that is to be proxied (hereafter simply referred to as the target class) doesn’t implement any interfaces, then a CGLIB-based proxy will be created. This is the easiest scenario, because JDK proxies are interface based, and no interfaces means JDK proxying isn’t even possible. One simply plugs in the target bean, and specifies the list of interceptors via the interceptorNames property. Note that a CGLIB-based proxy will be created even if the proxyTargetClass property of the ProxyFactoryBean has been set to false. (Obviously this makes no sense, and is best removed from the bean definition because it is at best redundant, and at worst confusing.)

If the target class implements one (or more) interfaces, then the type of proxy that is created depends on the configuration of the ProxyFactoryBean.

If the proxyTargetClass property of the ProxyFactoryBean has been set to true, then a CGLIB-based proxy will be created. This makes sense, and is in keeping with the principle of least surprise. Even if the proxyInterfaces property of the ProxyFactoryBean has been set to one or more fully qualified interface names, the fact that the proxyTargetClass property is set to true will cause CGLIB-based proxying to be in effect.

If the proxyInterfaces property of the ProxyFactoryBean has been set to one or more fully qualified interface names, then a JDK-based proxy will be created. The created proxy will implement all of the interfaces that were specified in the proxyInterfaces property; if the target class happens to implement a whole lot more interfaces than those specified in the proxyInterfaces property, that is all well and good but those additional interfaces will not be implemented by the returned proxy.

If the proxyInterfaces property of the ProxyFactoryBean has not been set, but the target class does implement one (or more) interfaces, then the ProxyFactoryBean will auto-detect the fact that the target class does actually implement at least one interface, and a JDK-based proxy will be created. The interfaces that are actually proxied will be all of the interfaces that the target class implements; in effect, this is the same as simply supplying a list of each and every interface that the target class implements to the proxyInterfaces property. However, it is significantly less work, and less prone to typos.

6.5.4. Proxying interfaces

Let’s look at a simple example of ProxyFactoryBean in action. This example involves:

A target bean that will be proxied. This is the "personTarget" bean definition in the example below.

An Advisor and an Interceptor used to provide advice.

An AOP proxy bean definition specifying the target object (the personTarget bean) and the interfaces to proxy, along with the advices to apply.

<bean id="personTarget" class="com.mycompany.PersonImpl">

<property name="name" value="Tony"/>

<property name="age" value="51"/>

</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">

<property name="someProperty" value="Custom string property value"/>

</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor">

</bean>

<bean id="person"

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="proxyInterfaces" value="com.mycompany.Person"/>

<property name="target" ref="personTarget"/>

<property name="interceptorNames">

<list>

<value>myAdvisor</value>

<value>debugInterceptor</value>

</list>

</property>

</bean>

Note that the interceptorNames property takes a list of String: the bean names of the interceptor or advisors in the current factory. Advisors, interceptors, before, after returning and throws advice objects can be used. The ordering of advisors is significant.

You might be wondering why the list doesn’t hold bean references. The reason for this is that if the ProxyFactoryBean’s singleton property is set to false, it must be able to return independent proxy instances. If any of the advisors is itself a prototype, an independent instance would need to be returned, so it’s necessary to be able to obtain an instance of the prototype from the factory; holding a reference isn’t sufficient.

The "person" bean definition above can be used in place of a Person implementation, as follows:

Person person = (Person) factory.getBean("person");

Other beans in the same IoC context can express a strongly typed dependency on it, as with an ordinary Java object:

<bean id="personUser" class="com.mycompany.PersonUser">

<property name="person"><ref bean="person"/></property>

</bean>

The PersonUser class in this example would expose a property of type Person. As far as it’s concerned, the AOP proxy can be used transparently in place of a "real" person implementation. However, its class would be a dynamic proxy class. It would be possible to cast it to the Advised interface (discussed below).

It’s possible to conceal the distinction between target and proxy using an anonymous inner bean, as follows. Only the ProxyFactoryBean definition is different; the advice is included only for completeness:

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">

<property name="someProperty" value="Custom string property value"/>

</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="proxyInterfaces" value="com.mycompany.Person"/>

<!-- Use inner bean, not local reference to target -->

<property name="target">

<bean class="com.mycompany.PersonImpl">

<property name="name" value="Tony"/>

<property name="age" value="51"/>

</bean>

</property>

<property name="interceptorNames">

<list>

<value>myAdvisor</value>

<value>debugInterceptor</value>

</list>

</property>

</bean>

This has the advantage that there’s only one object of type Person: useful if we want to prevent users of the application context from obtaining a reference to the un-advised object, or need to avoid any ambiguity with Spring IoC autowiring. There’s also arguably an advantage in that the ProxyFactoryBean definition is self-contained. However, there are times when being able to obtain the un-advised target from the factory might actually be an advantage: for example, in certain test scenarios.

6.5.5. Proxying classes

What if you need to proxy a class, rather than one or more interfaces?

Imagine that in our example above, there was no Person interface: we needed to advise a class called Person that didn’t implement any business interface. In this case, you can configure Spring to use CGLIB proxying, rather than dynamic proxies. Simply set the proxyTargetClass property on the ProxyFactoryBean above to true. While it’s best to program to interfaces, rather than classes, the ability to advise classes that don’t implement interfaces can be useful when working with legacy code. (In general, Spring isn’t prescriptive. While it makes it easy to apply good practices, it avoids forcing a particular approach.)

If you want to, you can force the use of CGLIB in any case, even if you do have interfaces.

CGLIB proxying works by generating a subclass of the target class at runtime. Spring configures this generated subclass to delegate method calls to the original target: the subclass is used to implement the Decorator pattern, weaving in the advice.

CGLIB proxying should generally be transparent to users. However, there are some issues to consider:

Final methods can’t be advised, as they can’t be overridden.

There is no need to add CGLIB to your classpath. As of Spring 3.2, CGLIB is repackaged and included in the spring-core JAR. In other words, CGLIB-based AOP will work "out of the box" just as do JDK dynamic proxies.

There’s little performance difference between CGLIB proxying and dynamic proxies. As of Spring 1.0, dynamic proxies are slightly faster. However, this may change in the future. Performance should not be a decisive consideration in this case.

6.5.6. Using 'global' advisors

By appending an asterisk to an interceptor name, all advisors with bean names matching the part before the asterisk, will be added to the advisor chain. This can come in handy if you need to add a standard set of 'global' advisors:

<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="target" ref="service"/>

<property name="interceptorNames">

<list>

<value>global\*</value>

</list>

</property>

</bean>

<bean id="global\_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>

<bean id="global\_performance" class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>

6.6. Concise proxy definitions

Especially when defining transactional proxies, you may end up with many similar proxy definitions. The use of parent and child bean definitions, along with inner bean definitions, can result in much cleaner and more concise proxy definitions.

First a parent, template, bean definition is created for the proxy:

<bean id="txProxyTemplate" abstract="true"

class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">

<property name="transactionManager" ref="transactionManager"/>

<property name="transactionAttributes">

<props>

<prop key="\*">PROPAGATION\_REQUIRED</prop>

</props>

</property>

</bean>

This will never be instantiated itself, so may actually be incomplete. Then each proxy which needs to be created is just a child bean definition, which wraps the target of the proxy as an inner bean definition, since the target will never be used on its own anyway.

<bean id="myService" parent="txProxyTemplate">

<property name="target">

<bean class="org.springframework.samples.MyServiceImpl">

</bean>

</property>

</bean>

It is of course possible to override properties from the parent template, such as in this case, the transaction propagation settings:

<bean id="mySpecialService" parent="txProxyTemplate">

<property name="target">

<bean class="org.springframework.samples.MySpecialServiceImpl">

</bean>

</property>

<property name="transactionAttributes">

<props>

<prop key="get\*">PROPAGATION\_REQUIRED,readOnly</prop>

<prop key="find\*">PROPAGATION\_REQUIRED,readOnly</prop>

<prop key="load\*">PROPAGATION\_REQUIRED,readOnly</prop>

<prop key="store\*">PROPAGATION\_REQUIRED</prop>

</props>

</property>

</bean>

Note that in the example above, we have explicitly marked the parent bean definition as abstract by using the abstract attribute, as described previously, so that it may not actually ever be instantiated. Application contexts (but not simple bean factories) will by default pre-instantiate all singletons. It is therefore important (at least for singleton beans) that if you have a (parent) bean definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the abstract attribute to true, otherwise the application context will actually try to pre-instantiate it.

6.7. Creating AOP proxies programmatically with the ProxyFactory

It’s easy to create AOP proxies programmatically using Spring. This enables you to use Spring AOP without dependency on Spring IoC.

The following listing shows creation of a proxy for a target object, with one interceptor and one advisor. The interfaces implemented by the target object will automatically be proxied:

ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);

factory.addAdvice(myMethodInterceptor);

factory.addAdvisor(myAdvisor);

MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();

The first step is to construct an object of type org.springframework.aop.framework.ProxyFactory. You can create this with a target object, as in the above example, or specify the interfaces to be proxied in an alternate constructor.

You can add advices (with interceptors as a specialized kind of advice) and/or advisors, and manipulate them for the life of the ProxyFactory. If you add an IntroductionInterceptionAroundAdvisor, you can cause the proxy to implement additional interfaces.

There are also convenience methods on ProxyFactory (inherited from AdvisedSupport) which allow you to add other advice types such as before and throws advice. AdvisedSupport is the superclass of both ProxyFactory and ProxyFactoryBean.

Integrating AOP proxy creation with the IoC framework is best practice in most applications. We recommend that you externalize configuration from Java code with AOP, as in general.

6.8. Manipulating advised objects

However you create AOP proxies, you can manipulate them using the org.springframework.aop.framework.Advised interface. Any AOP proxy can be cast to this interface, whichever other interfaces it implements. This interface includes the following methods:

Advisor[] getAdvisors();

void addAdvice(Advice advice) throws AopConfigException;

void addAdvice(int pos, Advice advice) throws AopConfigException;

void addAdvisor(Advisor advisor) throws AopConfigException;

void addAdvisor(int pos, Advisor advisor) throws AopConfigException;

int indexOf(Advisor advisor);

boolean removeAdvisor(Advisor advisor) throws AopConfigException;

void removeAdvisor(int index) throws AopConfigException;

boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException;

boolean isFrozen();

The getAdvisors() method will return an Advisor for every advisor, interceptor or other advice type that has been added to the factory. If you added an Advisor, the returned advisor at this index will be the object that you added. If you added an interceptor or other advice type, Spring will have wrapped this in an advisor with a pointcut that always returns true. Thus if you added a MethodInterceptor, the advisor returned for this index will be an DefaultPointcutAdvisor returning your MethodInterceptor and a pointcut that matches all classes and methods.

The addAdvisor() methods can be used to add any Advisor. Usually the advisor holding pointcut and advice will be the generic DefaultPointcutAdvisor, which can be used with any advice or pointcut (but not for introductions).

By default, it’s possible to add or remove advisors or interceptors even once a proxy has been created. The only restriction is that it’s impossible to add or remove an introduction advisor, as existing proxies from the factory will not show the interface change. (You can obtain a new proxy from the factory to avoid this problem.)

A simple example of casting an AOP proxy to the Advised interface and examining and manipulating its advice:

Advised advised = (Advised) myObject;

Advisor[] advisors = advised.getAdvisors();

int oldAdvisorCount = advisors.length;

System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut

// Will match all proxied methods

// Can use for interceptors, before, after returning or throws advice

advised.addAdvice(new DebugInterceptor());

// Add selective advice using a pointcut

advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors", oldAdvisorCount + 2, advised.getAdvisors().length);

It’s questionable whether it’s advisable (no pun intended) to modify advice on a business object in production, although there are no doubt legitimate usage cases. However, it can be very useful in development: for example, in tests. I have sometimes found it very useful to be able to add test code in the form of an interceptor or other advice, getting inside a method invocation I want to test. (For example, the advice can get inside a transaction created for that method: for example, to run SQL to check that a database was correctly updated, before marking the transaction for roll back.)

Depending on how you created the proxy, you can usually set a frozen flag, in which case the Advised isFrozen() method will return true, and any attempts to modify advice through addition or removal will result in an AopConfigException. The ability to freeze the state of an advised object is useful in some cases, for example, to prevent calling code removing a security interceptor. It may also be used in Spring 1.1 to allow aggressive optimization if runtime advice modification is known not to be required.

6.9. Using the "auto-proxy" facility

So far we’ve considered explicit creation of AOP proxies using a ProxyFactoryBean or similar factory bean.

Spring also allows us to use "auto-proxy" bean definitions, which can automatically proxy selected bean definitions. This is built on Spring "bean post processor" infrastructure, which enables modification of any bean definition as the container loads.

In this model, you set up some special bean definitions in your XML bean definition file to configure the auto proxy infrastructure. This allows you just to declare the targets eligible for auto-proxying: you don’t need to use ProxyFactoryBean.

There are two ways to do this:

Using an auto-proxy creator that refers to specific beans in the current context.

A special case of auto-proxy creation that deserves to be considered separately; auto-proxy creation driven by source-level metadata attributes.

6.9.1. Autoproxy bean definitions

The org.springframework.aop.framework.autoproxy package provides the following standard auto-proxy creators.

BeanNameAutoProxyCreator

The BeanNameAutoProxyCreator class is a BeanPostProcessor that automatically creates AOP proxies for beans with names matching literal values or wildcards.

<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">

<property name="beanNames" value="jdk\*,onlyJdk"/>

<property name="interceptorNames">

<list>

<value>myInterceptor</value>

</list>

</property>

</bean>

As with ProxyFactoryBean, there is an interceptorNames property rather than a list of interceptors, to allow correct behavior for prototype advisors. Named "interceptors" can be advisors or any advice type.

As with auto proxying in general, the main point of using BeanNameAutoProxyCreator is to apply the same configuration consistently to multiple objects, with minimal volume of configuration. It is a popular choice for applying declarative transactions to multiple objects.

Bean definitions whose names match, such as "jdkMyBean" and "onlyJdk" in the above example, are plain old bean definitions with the target class. An AOP proxy will be created automatically by the BeanNameAutoProxyCreator. The same advice will be applied to all matching beans. Note that if advisors are used (rather than the interceptor in the above example), the pointcuts may apply differently to different beans.

DefaultAdvisorAutoProxyCreator

A more general and extremely powerful auto proxy creator is DefaultAdvisorAutoProxyCreator. This will automagically apply eligible advisors in the current context, without the need to include specific bean names in the auto-proxy advisor’s bean definition. It offers the same merit of consistent configuration and avoidance of duplication as BeanNameAutoProxyCreator.

Using this mechanism involves:

Specifying a DefaultAdvisorAutoProxyCreator bean definition.

Specifying any number of Advisors in the same or related contexts. Note that these must be Advisors, not just interceptors or other advices. This is necessary because there must be a pointcut to evaluate, to check the eligibility of each advice to candidate bean definitions.

The DefaultAdvisorAutoProxyCreator will automatically evaluate the pointcut contained in each advisor, to see what (if any) advice it should apply to each business object (such as "businessObject1" and "businessObject2" in the example).

This means that any number of advisors can be applied automatically to each business object. If no pointcut in any of the advisors matches any method in a business object, the object will not be proxied. As bean definitions are added for new business objects, they will automatically be proxied if necessary.

Autoproxying in general has the advantage of making it impossible for callers or dependencies to obtain an un-advised object. Calling getBean("businessObject1") on this ApplicationContext will return an AOP proxy, not the target business object. (The "inner bean" idiom shown earlier also offers this benefit.)

<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">

<property name="transactionInterceptor" ref="transactionInterceptor"/>

</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">

<!-- Properties omitted -->

</bean>

<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>

The DefaultAdvisorAutoProxyCreator is very useful if you want to apply the same advice consistently to many business objects. Once the infrastructure definitions are in place, you can simply add new business objects without including specific proxy configuration. You can also drop in additional aspects very easily - for example, tracing or performance monitoring aspects - with minimal change to configuration.

The DefaultAdvisorAutoProxyCreator offers support for filtering (using a naming convention so that only certain advisors are evaluated, allowing use of multiple, differently configured, AdvisorAutoProxyCreators in the same factory) and ordering. Advisors can implement the org.springframework.core.Ordered interface to ensure correct ordering if this is an issue. The TransactionAttributeSourceAdvisor used in the above example has a configurable order value; the default setting is unordered.

6.10. Using TargetSources

Spring offers the concept of a TargetSource, expressed in the org.springframework.aop.TargetSource interface. This interface is responsible for returning the "target object" implementing the join point. The TargetSource implementation is asked for a target instance each time the AOP proxy handles a method invocation.

Developers using Spring AOP don’t normally need to work directly with TargetSources, but this provides a powerful means of supporting pooling, hot swappable and other sophisticated targets. For example, a pooling TargetSource can return a different target instance for each invocation, using a pool to manage instances.

If you do not specify a TargetSource, a default implementation is used that wraps a local object. The same target is returned for each invocation (as you would expect).

Let’s look at the standard target sources provided with Spring, and how you can use them.

When using a custom target source, your target will usually need to be a prototype rather than a singleton bean definition. This allows Spring to create a new target instance when required.

6.10.1. Hot swappable target sources

The org.springframework.aop.target.HotSwappableTargetSource exists to allow the target of an AOP proxy to be switched while allowing callers to keep their references to it.

Changing the target source’s target takes effect immediately. The HotSwappableTargetSource is threadsafe.

You can change the target via the swap() method on HotSwappableTargetSource as follows:

HotSwappableTargetSource swapper = (HotSwappableTargetSource) beanFactory.getBean("swapper");

Object oldTarget = swapper.swap(newTarget);

The XML definitions required look as follows:

<bean id="initialTarget" class="mycompany.OldTarget"/>

<bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">

<constructor-arg ref="initialTarget"/>

</bean>

<bean id="swappable" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="targetSource" ref="swapper"/>

</bean>

The above swap() call changes the target of the swappable bean. Clients who hold a reference to that bean will be unaware of the change, but will immediately start hitting the new target.

Although this example doesn’t add any advice - and it’s not necessary to add advice to use a TargetSource - of course any TargetSource can be used in conjunction with arbitrary advice.

6.10.2. Pooling target sources

Using a pooling target source provides a similar programming model to stateless session EJBs, in which a pool of identical instances is maintained, with method invocations going to free objects in the pool.

A crucial difference between Spring pooling and SLSB pooling is that Spring pooling can be applied to any POJO. As with Spring in general, this service can be applied in a non-invasive way.

Spring provides out-of-the-box support for Commons Pool 2.2, which provides a fairly efficient pooling implementation. You’ll need the commons-pool Jar on your application’s classpath to use this feature. It’s also possible to subclass org.springframework.aop.target.AbstractPoolingTargetSource to support any other pooling API.

Commons Pool 1.5+ is also supported but deprecated as of Spring Framework 4.2.

Sample configuration is shown below:

<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject"

scope="prototype">

... properties omitted

</bean>

<bean id="poolTargetSource" class="org.springframework.aop.target.CommonsPool2TargetSource">

<property name="targetBeanName" value="businessObjectTarget"/>

<property name="maxSize" value="25"/>

</bean>

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="targetSource" ref="poolTargetSource"/>

<property name="interceptorNames" value="myInterceptor"/>

</bean>

Note that the target object - "businessObjectTarget" in the example - must be a prototype. This allows the PoolingTargetSource implementation to create new instances of the target to grow the pool as necessary. See the javadocs of AbstractPoolingTargetSource and the concrete subclass you wish to use for information about its properties: "maxSize" is the most basic, and always guaranteed to be present.

In this case, "myInterceptor" is the name of an interceptor that would need to be defined in the same IoC context. However, it isn’t necessary to specify interceptors to use pooling. If you want only pooling, and no other advice, don’t set the interceptorNames property at all.

It’s possible to configure Spring so as to be able to cast any pooled object to the org.springframework.aop.target.PoolingConfig interface, which exposes information about the configuration and current size of the pool through an introduction. You’ll need to define an advisor like this:

<bean id="poolConfigAdvisor" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">

<property name="targetObject" ref="poolTargetSource"/>

<property name="targetMethod" value="getPoolingConfigMixin"/>

</bean>

This advisor is obtained by calling a convenience method on the AbstractPoolingTargetSource class, hence the use of MethodInvokingFactoryBean. This advisor’s name ("poolConfigAdvisor" here) must be in the list of interceptors names in the ProxyFactoryBean exposing the pooled object.

The cast will look as follows:

PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");

System.out.println("Max pool size is " + conf.getMaxSize());

Pooling stateless service objects is not usually necessary. We don’t believe it should be the default choice, as most stateless objects are naturally thread safe, and instance pooling is problematic if resources are cached.

Simpler pooling is available using auto-proxying. It’s possible to set the TargetSources used by any auto-proxy creator.

6.10.3. Prototype target sources

Setting up a "prototype" target source is similar to a pooling TargetSource. In this case, a new instance of the target will be created on every method invocation. Although the cost of creating a new object isn’t high in a modern JVM, the cost of wiring up the new object (satisfying its IoC dependencies) may be more expensive. Thus you shouldn’t use this approach without very good reason.

To do this, you could modify the poolTargetSource definition shown above as follows. (I’ve also changed the name, for clarity.)

<bean id="prototypeTargetSource" class="org.springframework.aop.target.PrototypeTargetSource">

<property name="targetBeanName" ref="businessObjectTarget"/>

</bean>

There’s only one property: the name of the target bean. Inheritance is used in the TargetSource implementations to ensure consistent naming. As with the pooling target source, the target bean must be a prototype bean definition.

6.10.4. ThreadLocal target sources

ThreadLocal target sources are useful if you need an object to be created for each incoming request (per thread that is). The concept of a ThreadLocal provide a JDK-wide facility to transparently store resource alongside a thread. Setting up a ThreadLocalTargetSource is pretty much the same as was explained for the other types of target source:

<bean id="threadlocalTargetSource" class="org.springframework.aop.target.ThreadLocalTargetSource">

<property name="targetBeanName" value="businessObjectTarget"/>

</bean>

ThreadLocals come with serious issues (potentially resulting in memory leaks) when incorrectly using them in a multi-threaded and multi-classloader environments. One should always consider wrapping a threadlocal in some other class and never directly use the ThreadLocal itself (except of course in the wrapper class). Also, one should always remember to correctly set and unset (where the latter simply involved a call to ThreadLocal.set(null)) the resource local to the thread. Unsetting should be done in any case since not unsetting it might result in problematic behavior. Spring’s ThreadLocal support does this for you and should always be considered in favor of using ThreadLocals without other proper handling code.

6.11. Defining new Advice types

Spring AOP is designed to be extensible. While the interception implementation strategy is presently used internally, it is possible to support arbitrary advice types in addition to the out-of-the-box interception around advice, before, throws advice and after returning advice.

The org.springframework.aop.framework.adapter package is an SPI package allowing support for new custom advice types to be added without changing the core framework. The only constraint on a custom Advice type is that it must implement the org.aopalliance.aop.Advice marker interface.

Please refer to the org.springframework.aop.framework.adapter javadocs for further information.

7. Null-safety

Although Java does not allow to express null-safety with its type system, Spring Framework now provides following annotations in the org.springframework.lang package to declare nullability of APIs and fields:

@NonNull annotation where specific parameter, return value or field cannot be null (not needed on parameter and return value where @NonNullApi and @NonNullFields apply) .

@Nullable annotation where specific parameter, return value or field can be null.

@NonNullApi annotation at package level declares non-null as the default behavior for parameters and return values.

@NonNullFields annotation at package level declares non-null as the default behavior for fields.

Spring Framework leverages itself these annotations, but they can also be used in any Spring based Java project to declare null-safe APIs and optionally null-safe fields. Generic type arguments, varargs and array elements nullability are not supported yet, but should be in an upcoming release, see SPR-15942 for up-to-date information. Nullability declaration are expected to be fine-tuned between Spring Framework release, including minor ones. Nullability of types used inside method bodies is outside of the scope of this feature.

Libraries like Reactor or Spring Data provide null-safe APIs leveraging this feature.

7.1. Use cases

In addition to providing an explicit declaration for Spring Framework API nullability, these annotation can be used by IDE (such as IDEA or Eclipse) to provide useful warnings to Java developers related to null-safety in order to avoid NullPointerException at runtime.

They are also used to make Spring API null-safe in Kotlin projects since Kotlin natively supports null-safety. More details are available in Kotlin support documentation.

7.2. JSR 305 meta-annotations

Spring annotations are meta-annotated with JSR 305 annotations (a dormant but widely spread JSR). JSR 305 meta-annotations allows tooling vendors like IDEA or Kotlin to provide null-safety support in a generic way, without having to hard-code support for Spring annotations.

It is not necessary nor recommended to add JSR 305 dependency in project classpath to take advantage of Spring null-safe API. Only projects like Spring-based libraries using null-safety annotations in their codebase should add com.google.code.findbugs:jsr305:3.0.2 with compileOnly Gradle configuration or Maven provided scope to avoid compile warnings.

8. Data Buffers and Codecs

8.1. Introduction

The DataBuffer interface defines an abstraction over byte buffers. The main reason for introducing it, and not use the standard java.nio.ByteBuffer instead, is Netty. Netty does not use ByteBuffer, but instead offers ByteBuf as an alternative. Spring’s DataBuffer is a simple abstraction over ByteBuf that can also be used on non-Netty platforms (i.e. Servlet 3.1+).

8.2. DataBufferFactory

The DataBufferFactory offers functionality to allocate new data buffers, as well as to wrap existing data. The allocate methods allocate a new data buffer, with a default or given capacity. Though DataBuffer implementation grow and shrink on demand, it is more efficient to give the capacity upfront, if known. The wrap methods decorate an existing ByteBuffer or byte array. Wrapping does not involve allocation: it simply decorates the given data with a DataBuffer implementation.

There are two implementation of DataBufferFactory: the NettyDataBufferFactory which is meant to be used on Netty platforms, such as Reactor Netty. The other implementation, the DefaultDataBufferFactory, is used on other platforms, such as Servlet 3.1+ servers.

8.3. The DataBuffer interface

The DataBuffer interface is similar to ByteBuffer, but offers a number of advantages. Similar to Netty’s ByteBuf, the DataBuffer abstraction offers independent read and write positions. This is different from the JDK’s ByteBuffer, which only exposes one position for both reading and writing, and a separate flip() operation to switch between the two I/O operations. In general, the following invariant holds for the read position, write position, and the capacity:

0 <= read position <= write position <= capacity

When reading bytes from the DataBuffer, the read position is automatically updated in accordance with the amount of data read from the buffer. Similarly, when writing bytes to the DataBuffer, the write position is updated with the amount of data written to the buffer. Also, when writing data, the capacity of a DataBuffer is automatically expanded, just like StringBuilder, ArrayList, and similar types.

Besides the reading and writing functionality mentioned above, the DataBuffer also has methods to view a (slice of a) buffer as ByteBuffer, InputStream, or OutputStream. Additionally, it offers methods to determine the index of a given byte.

There are two implementation of DataBuffer: the NettyDataBuffer which is meant to be used on Netty platforms, such as Reactor Netty. The other implementation, the DefaultDataBuffer, is used on other platforms, such as Servlet 3.1+ servers.

8.3.1. PooledDataBuffer

The PooledDataBuffer is an extension to DataBuffer that adds methods for reference counting. The retain method increases the reference count by one. The release method decreases the count by one, and releases the buffer’s memory when the count reaches 0. Both of these methods are related to reference counting, a mechanism that is explained below.

Note that DataBufferUtils offers useful utility methods for releasing and retaining pooled data buffers. These methods take a plain DataBuffer as parameter, but only call retain or release if the passed data buffer is an instance of PooledDataBuffer.

Reference Counting

Reference counting is not a common technique in Java; it is much more common in other programming languages such as Object C and C++. In and of itself, reference counting is not complex: it basically involves tracking the number of references that apply to an object. The reference count of a PooledDataBuffer starts at 1, is incremented by calling retain, and decremented by calling release. As long as the buffer’s reference count is larger than 0 the buffer will not be released. When the number decreases to 0, the instance will be released. In practice, this means that the reserved memory captured by the buffer will be returned back to the memory pool, ready to be used for future allocations.

In general, the last component to access a DataBuffer is responsible for releasing it. Withing Spring, there are two sorts of components that release buffers: decoders and transports. Decoders are responsible for transforming a stream of buffers into other types (see Codecs below), and transports are responsible for sending buffers across a network boundary, typically as an HTTP message. This means that if you allocate data buffers for the purpose of putting them into an outbound HTTP message (i.e. client-side request or server-side response), they do not have to be released. The other consequence of this rule is that if you allocate data buffers that do not end up in the body, for instance because of a thrown exception, you will have to release them yourself. The following snippet shows a typical DataBuffer usage scenario when dealing with methods that throw exceptions:

DataBufferFactory factory = ...

DataBuffer buffer = factory.allocateBuffer();

boolean release = true;

try {

writeDataToBuffer(buffer);

putBufferInHttpBody(buffer);

release = false;

}

finally {

if (release) {

DataBufferUtils.release(buffer);

}

}

private void writeDataToBuffer(DataBuffer buffer) throws IOException {

...

}

A new buffer is allocated.

A boolean flag indicates whether the allocated buffer should be released.

This example method loads data into the buffer. Note that the method can throw an IOException, and therefore a finally block to release the buffer is required.

If no exception occurred, we switch the release flag to false as the buffer will now be released as part of sending the HTTP body across the wire.

If an exception did occur, the flag is still set to true, and the buffer will be released here.

8.3.2. DataBufferUtils

DataBufferUtils contains various utility methods that operate on data buffers. It contains methods for reading a Flux of DataBuffer objects from an InputStream or NIO Channel, and methods for writing a data buffer Flux to an OutputStream or Channel. DataBufferUtils also exposes retain and release methods that operate on plain DataBuffer instances (so that casting to a PooledDataBuffer is not required).

Additionally, DataBufferUtils exposes compose, which merges a stream of data buffers into one. For instance, this method can be used to convert the entire HTTP body into a single buffer (and from that, a String, or InputStream). This is particularly useful when dealing with older, blocking APIs. Note, however, that this puts the entire body in memory, and therefore uses more memory than a pure streaming solution would.

Codecs

The org.springframework.core.codec package contains the two main abstractions for converting a stream of bytes into a stream of objects, or vice-versa. The Encoder is a strategy interface that encodes a stream of objects into an output stream of data buffers. The Decoder does the reverse: it turns a stream of data buffers into a stream of objects. Note that a decoder instance needs to consider reference counting.

Spring comes with a wide array of default codecs, capable of converting from/to String, ByteBuffer, byte arrays, and also codecs that support marshalling libraries such as JAXB and Jackson (with Jackson 2.9+ support for non-blocking parsing). Withing the context of Spring WebFlux, codecs are used to convert the request body into a @RequestMapping parameter, or to convert the return type into the response body that is sent back to the client. The default codecs are configured in the WebFluxConfigurationSupport class, and can easily be changed by overriding the configureHttpMessageCodecs when inheriting from that class. For more information about using codecs in WebFlux, see this section.

9. Appendix

9.1. XML Schemas

This part of the appendix lists XML schemas related to the core container.

9.1.1. The util schema

As the name implies, the util tags deal with common, utility configuration issues, such as configuring collections, referencing constants, and suchlike. To use the tags in the util schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the snippet below references the correct schema so that the tags in the util namespace are available to you.

<?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:util="http://www.springframework.org/schema/util" xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd"> <!-- bean definitions here -->

</beans>

<util:constant/>

Before…​

<bean id="..." class="...">

<property name="isolation">

<bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />

</property>

</bean>

The above configuration uses a Spring FactoryBean implementation, the FieldRetrievingFactoryBean, to set the value of the isolation property on a bean to the value of the java.sql.Connection.TRANSACTION\_SERIALIZABLE constant. This is all well and good, but it is a tad verbose and (unnecessarily) exposes Spring’s internal plumbing to the end user.

The following XML Schema-based version is more concise and clearly expresses the developer’s intent ('inject this constant value'), and it just reads better.

<bean id="..." class="...">

<property name="isolation">

<util:constant static-field="java.sql.Connection.TRANSACTION\_SERIALIZABLE"/>

</property>

</bean>

Setting a bean property or constructor arg from a field value

FieldRetrievingFactoryBean is a FactoryBean which retrieves a static or non-static field value. It is typically used for retrieving public static final constants, which may then be used to set a property value or constructor arg for another bean.

Find below an example which shows how a static field is exposed, by using the staticField property:

<bean id="myField"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">

<property name="staticField" value="java.sql.Connection.TRANSACTION\_SERIALIZABLE"/>

</bean>

There is also a convenience usage form where the static field is specified as the bean name:

<bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>

This does mean that there is no longer any choice in what the bean id is (so any other bean that refers to it will also have to use this longer name), but this form is very concise to define, and very convenient to use as an inner bean since the id doesn’t have to be specified for the bean reference:

<bean id="..." class="...">

<property name="isolation">

<bean id="java.sql.Connection.TRANSACTION\_SERIALIZABLE"

class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />

</property>

</bean>

It is also possible to access a non-static (instance) field of another bean, as described in the API documentation for the FieldRetrievingFactoryBean class.

Injecting enum values into beans as either property or constructor arguments is very easy to do in Spring, in that you don’t actually have to do anything or know anything about the Spring internals (or even about classes such as the FieldRetrievingFactoryBean). Let’s look at an example to see how easy injecting an enum value is; consider this enum:

package javax.persistence;

public enum PersistenceContextType {

TRANSACTION,

EXTENDED

}

Now consider a setter of type PersistenceContextType:

package example;

public class Client {

private PersistenceContextType persistenceContextType;

public void setPersistenceContextType(PersistenceContextType type) {

this.persistenceContextType = type;

}

}

and the corresponding bean definition:

<bean class="example.Client">

<property name="persistenceContextType" value="TRANSACTION"/>

</bean>

<util:property-path/>

Before…​

<!-- target bean to be referenced by name -->

<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">

<property name="age" value="10"/>

<property name="spouse">

<bean class="org.springframework.beans.TestBean">

<property name="age" value="11"/>

</bean>

</property>

</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->

<bean id="testBean.age" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

The above configuration uses a Spring FactoryBean implementation, the PropertyPathFactoryBean, to create a bean (of type int) called testBean.age that has a value equal to the age property of the testBean bean.

After…​

<!-- target bean to be referenced by name -->

<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">

<property name="age" value="10"/>

<property name="spouse">

<bean class="org.springframework.beans.TestBean">

<property name="age" value="11"/>

</bean>

</property>

</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->

<util:property-path id="name" path="testBean.age"/>

The value of the path attribute of the <property-path/> tag follows the form beanName.beanProperty.

Using <util:property-path/> to set a bean property or constructor-argument

PropertyPathFactoryBean is a FactoryBean that evaluates a property path on a given target object. The target object can be specified directly or via a bean name. This value may then be used in another bean definition as a property value or constructor argument.

Here’s an example where a path is used against another bean, by name:

// target bean to be referenced by name

<bean id="person" class="org.springframework.beans.TestBean" scope="prototype">

<property name="age" value="10"/>

<property name="spouse">

<bean class="org.springframework.beans.TestBean">

<property name="age" value="11"/>

</bean>

</property>

</bean>

// will result in 11, which is the value of property 'spouse.age' of bean 'person'

<bean id="theAge"

class="org.springframework.beans.factory.config.PropertyPathFactoryBean">

<property name="targetBeanName" value="person"/>

<property name="propertyPath" value="spouse.age"/>

</bean>

In this example, a path is evaluated against an inner bean:

<!-- will result in 12, which is the value of property 'age' of the inner bean -->

<bean id="theAge"

class="org.springframework.beans.factory.config.PropertyPathFactoryBean">

<property name="targetObject">

<bean class="org.springframework.beans.TestBean">

<property name="age" value="12"/>

</bean>

</property>

<property name="propertyPath" value="age"/>

</bean>

There is also a shortcut form, where the bean name is the property path.

<!-- will result in 10, which is the value of property 'age' of bean 'person' -->

<bean id="person.age"

class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

This form does mean that there is no choice in the name of the bean. Any reference to it will also have to use the same id, which is the path. Of course, if used as an inner bean, there is no need to refer to it at all:

<bean id="..." class="...">

<property name="age">

<bean id="person.age"

class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>

</property>

</bean>

The result type may be specifically set in the actual definition. This is not necessary for most use cases, but can be of use for some. Please see the Javadocs for more info on this feature.

<util:properties/>

Before…​

<!-- creates a java.util.Properties instance with values loaded from the supplied location -->

<bean id="jdbcConfiguration" class="org.springframework.beans.factory.config.PropertiesFactoryBean">

<property name="location" value="classpath:com/foo/jdbc-production.properties"/>

</bean>

The above configuration uses a Spring FactoryBean implementation, the PropertiesFactoryBean, to instantiate a java.util.Properties instance with values loaded from the supplied Resource location).

After…​

<!-- creates a java.util.Properties instance with values loaded from the supplied location -->

<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-production.properties"/>

<util:list/>

Before…​

<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->

<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">

<property name="sourceList">

<list>

<value>pechorin@hero.org</value>

<value>raskolnikov@slums.org</value>

<value>stavrogin@gov.org</value>

<value>porfiry@gov.org</value>

</list>

</property>

</bean>

The above configuration uses a Spring FactoryBean implementation, the ListFactoryBean, to create a java.util.List instance initialized with values taken from the supplied sourceList.

After…​

<!-- creates a java.util.List instance with the supplied values -->

<util:list id="emails">

<value>pechorin@hero.org</value>

<value>raskolnikov@slums.org</value>

<value>stavrogin@gov.org</value>

<value>porfiry@gov.org</value>

</util:list>

You can also explicitly control the exact type of List that will be instantiated and populated via the use of the list-class attribute on the <util:list/> element. For example, if we really need a java.util.LinkedList to be instantiated, we could use the following configuration:

<util:list id="emails" list-class="java.util.LinkedList">

<value>jackshaftoe@vagabond.org</value>

<value>eliza@thinkingmanscrumpet.org</value>

<value>vanhoek@pirate.org</value>

<value>d'Arcachon@nemesis.org</value>

</util:list>

If no list-class attribute is supplied, a List implementation will be chosen by the container.

<util:map/>

Before…​

<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->

<bean id="emails" class="org.springframework.beans.factory.config.MapFactoryBean">

<property name="sourceMap">

<map>

<entry key="pechorin" value="pechorin@hero.org"/>

<entry key="raskolnikov" value="raskolnikov@slums.org"/>

<entry key="stavrogin" value="stavrogin@gov.org"/>

<entry key="porfiry" value="porfiry@gov.org"/>

</map>

</property>

</bean>

The above configuration uses a Spring FactoryBean implementation, the MapFactoryBean, to create a java.util.Map instance initialized with key-value pairs taken from the supplied 'sourceMap'.

After…​

<!-- creates a java.util.Map instance with the supplied key-value pairs -->

<util:map id="emails">

<entry key="pechorin" value="pechorin@hero.org"/>

<entry key="raskolnikov" value="raskolnikov@slums.org"/>

<entry key="stavrogin" value="stavrogin@gov.org"/>

<entry key="porfiry" value="porfiry@gov.org"/>

</util:map>

You can also explicitly control the exact type of Map that will be instantiated and populated via the use of the 'map-class' attribute on the <util:map/> element. For example, if we really need a java.util.TreeMap to be instantiated, we could use the following configuration:

<util:map id="emails" map-class="java.util.TreeMap">

<entry key="pechorin" value="pechorin@hero.org"/>

<entry key="raskolnikov" value="raskolnikov@slums.org"/>

<entry key="stavrogin" value="stavrogin@gov.org"/>

<entry key="porfiry" value="porfiry@gov.org"/>

</util:map>

If no 'map-class' attribute is supplied, a Map implementation will be chosen by the container.

<util:set/>

Before…​

<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet' -->

<bean id="emails" class="org.springframework.beans.factory.config.SetFactoryBean">

<property name="sourceSet">

<set>

<value>pechorin@hero.org</value>

<value>raskolnikov@slums.org</value>

<value>stavrogin@gov.org</value>

<value>porfiry@gov.org</value>

</set>

</property>

</bean>

The above configuration uses a Spring FactoryBean implementation, the SetFactoryBean, to create a java.util.Set instance initialized with values taken from the supplied 'sourceSet'.

After…​

<!-- creates a java.util.Set instance with the supplied values -->

<util:set id="emails">

<value>pechorin@hero.org</value>

<value>raskolnikov@slums.org</value>

<value>stavrogin@gov.org</value>

<value>porfiry@gov.org</value>

</util:set>

You can also explicitly control the exact type of Set that will be instantiated and populated via the use of the 'set-class' attribute on the <util:set/> element. For example, if we really need a java.util.TreeSet to be instantiated, we could use the following configuration:

<util:set id="emails" set-class="java.util.TreeSet">

<value>pechorin@hero.org</value>

<value>raskolnikov@slums.org</value>

<value>stavrogin@gov.org</value>

<value>porfiry@gov.org</value>

</util:set>

If no 'set-class' attribute is supplied, a Set implementation will be chosen by the container.

9.1.2. The aop schema

The aop tags deal with configuring all things AOP in Spring: this includes Spring’s own proxy-based AOP framework and Spring’s integration with the AspectJ AOP framework. These tags are comprehensively covered in the chapter entitled Aspect Oriented Programming with Spring.

In the interest of completeness, to use the tags in the aop schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the aop namespace are available to you.

<?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:aop="http://www.springframework.org/schema/aop" xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd"> <!-- bean definitions here -->

</beans>

9.1.3. The context schema

The context tags deal with ApplicationContext configuration that relates to plumbing - that is, not usually beans that are important to an end-user but rather beans that do a lot of grunt work in Spring, such as BeanfactoryPostProcessors. The following snippet references the correct schema so that the tags in the context namespace are available to you.

<?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"> <!-- bean definitions here -->

</beans>

<property-placeholder/>

This element activates the replacement of ${…​} placeholders, resolved against the specified properties file (as a Spring resource location). This element is a convenience mechanism that sets up aPropertyPlaceholderConfigurer for you; if you need more control over the PropertyPlaceholderConfigurer, just define one yourself explicitly.

<annotation-config/>

Activates the Spring infrastructure for various annotations to be detected in bean classes: Spring’s @Required and @Autowired, as well as JSR 250’s @PostConstruct, @PreDestroy and @Resource (if available), and JPA’s @PersistenceContext and @PersistenceUnit (if available). Alternatively, you can choose to activate the individual BeanPostProcessors for those annotations explicitly.

This element does not activate processing of Spring’s @Transactional annotation. Use the <tx:annotation-driven/> element for that purpose.

<component-scan/>

This element is detailed in Annotation-based container configuration.

<load-time-weaver/>

This element is detailed in Load-time weaving with AspectJ in the Spring Framework.

<spring-configured/>

This element is detailed in Using AspectJ to dependency inject domain objects with Spring.

<mbean-export/>

This element is detailed in Configuring annotation based MBean export.

9.1.4. The beans schema

Last but not least we have the tags in the beans schema. These are the same tags that have been in Spring since the very dawn of the framework. Examples of the various tags in the beans schema are not shown here because they are quite comprehensively covered in Dependencies and configuration in detail (and indeed in that entire chapter).

Note that it is possible to add zero or more key / value pairs to <bean/> XML definitions. What, if anything, is done with this extra metadata is totally up to your own custom logic (and so is typically only of use if you are writing your own custom tags as described in the appendix entitled XML Schema Authoring).

Find below an example of the <meta/> tag in the context of a surrounding <bean/> (please note that without any logic to interpret it the metadata is effectively useless as-is).

<?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="foo" class="x.y.Foo">

<meta key="cacheName" value="foo"/>

<property name="name" value="Rick"/>

</bean>

</beans>

In the case of the above example, you would assume that there is some logic that will consume the bean definition and set up some caching infrastructure using the supplied metadata.

9.2. XML Schema Authoring

9.2.1. Introduction

Since version 2.0, Spring has featured a mechanism for schema-based extensions to the basic Spring XML format for defining and configuring beans. This section is devoted to detailing how you would go about writing your own custom XML bean definition parsers and integrating such parsers into the Spring IoC container.

To facilitate the authoring of configuration files using a schema-aware XML editor, Spring’s extensible XML configuration mechanism is based on XML Schema. If you are not familiar with Spring’s current XML configuration extensions that come with the standard Spring distribution, please first read the appendix entitled[xsd-config].

Creating new XML configuration extensions can be done by following these (relatively) simple steps:

Authoring an XML schema to describe your custom element(s).

Coding a custom NamespaceHandler implementation (this is an easy step, don’t worry).

Coding one or more BeanDefinitionParser implementations (this is where the real work is done).

Registering the above artifacts with Spring (this too is an easy step).

What follows is a description of each of these steps. For the example, we will create an XML extension (a custom XML element) that allows us to configure objects of the type SimpleDateFormat (from the java.text package) in an easy manner. When we are done, we will be able to define bean definitions of type SimpleDateFormat like this:

<myns:dateformat id="dateFormat"

pattern="yyyy-MM-dd HH:mm"

lenient="true"/>

(Don’t worry about the fact that this example is very simple; much more detailed examples follow afterwards. The intent in this first simple example is to walk you through the basic steps involved.)

9.2.2. Authoring the schema

Creating an XML configuration extension for use with Spring’s IoC container starts with authoring an XML Schema to describe the extension. What follows is the schema we’ll use to configure SimpleDateFormat objects.

<!-- myns.xsd (inside package org/springframework/samples/xml) -->

<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns="http://www.mycompany.com/schema/myns"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns:beans="http://www.springframework.org/schema/beans"

targetNamespace="http://www.mycompany.com/schema/myns"

elementFormDefault="qualified"

attributeFormDefault="unqualified">

<xsd:import namespace="http://www.springframework.org/schema/beans"/>

<xsd:element name="dateformat">

<xsd:complexType>

<xsd:complexContent>

<xsd:extension base="beans:identifiedType">

<xsd:attribute name="lenient" type="xsd:boolean"/>

<xsd:attribute name="pattern" type="xsd:string" use="required"/>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

</xsd:element>

</xsd:schema>

(The emphasized line contains an extension base for all tags that will be identifiable (meaning they have an id attribute that will be used as the bean identifier in the container). We are able to use this attribute because we imported the Spring-provided 'beans' namespace.)

The above schema will be used to configure SimpleDateFormat objects, directly in an XML application context file using the <myns:dateformat/> element.

<myns:dateformat id="dateFormat"

pattern="yyyy-MM-dd HH:mm"

lenient="true"/>

Note that after we’ve created the infrastructure classes, the above snippet of XML will essentially be exactly the same as the following XML snippet. In other words, we’re just creating a bean in the container, identified by the name 'dateFormat' of type SimpleDateFormat, with a couple of properties set.

<bean id="dateFormat" class="java.text.SimpleDateFormat">

<constructor-arg value="yyyy-HH-dd HH:mm"/>

<property name="lenient" value="true"/>

</bean>

The schema-based approach to creating configuration format allows for tight integration with an IDE that has a schema-aware XML editor. Using a properly authored schema, you can use autocompletion to have a user choose between several configuration options defined in the enumeration.

9.2.3. Coding a NamespaceHandler

In addition to the schema, we need a NamespaceHandler that will parse all elements of this specific namespace Spring encounters while parsing configuration files. The NamespaceHandler should in our case take care of the parsing of the myns:dateformat element.

The NamespaceHandler interface is pretty simple in that it features just three methods:

init() - allows for initialization of the NamespaceHandler and will be called by Spring before the handler is used

BeanDefinition parse(Element, ParserContext) - called when Spring encounters a top-level element (not nested inside a bean definition or a different namespace). This method can register bean definitions itself and/or return a bean definition.

BeanDefinitionHolder decorate(Node, BeanDefinitionHolder, ParserContext) - called when Spring encounters an attribute or nested element of a different namespace. The decoration of one or more bean definitions is used for example with the out-of-the-box scopes Spring supports. We’ll start by highlighting a simple example, without using decoration, after which we will show decoration in a somewhat more advanced example.

Although it is perfectly possible to code your own NamespaceHandler for the entire namespace (and hence provide code that parses each and every element in the namespace), it is often the case that each top-level XML element in a Spring XML configuration file results in a single bean definition (as in our case, where a single <myns:dateformat/> element results in a single SimpleDateFormat bean definition). Spring features a number of convenience classes that support this scenario. In this example, we’ll make use the NamespaceHandlerSupport class:

package org.springframework.samples.xml;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class MyNamespaceHandler extends NamespaceHandlerSupport {

public void init() {

registerBeanDefinitionParser("dateformat", new SimpleDateFormatBeanDefinitionParser());

}

}

The observant reader will notice that there isn’t actually a whole lot of parsing logic in this class. Indeed…​ the NamespaceHandlerSupport class has a built in notion of delegation. It supports the registration of any number of BeanDefinitionParser instances, to which it will delegate to when it needs to parse an element in its namespace. This clean separation of concerns allows a NamespaceHandler to handle the orchestration of the parsing of all of the custom elements in its namespace, while delegating to BeanDefinitionParsers to do the grunt work of the XML parsing; this means that each BeanDefinitionParser will contain just the logic for parsing a single custom element, as we can see in the next step

9.2.4. BeanDefinitionParser

A BeanDefinitionParser will be used if the NamespaceHandler encounters an XML element of the type that has been mapped to the specific bean definition parser (which is 'dateformat' in this case). In other words, the BeanDefinitionParser is responsible for parsing one distinct top-level XML element defined in the schema. In the parser, we’ll have access to the XML element (and thus its subelements too) so that we can parse our custom XML content, as can be seen in the following example:

package org.springframework.samples.xml;

import org.springframework.beans.factory.support.BeanDefinitionBuilder;

import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;

import org.springframework.util.StringUtils;

import org.w3c.dom.Element;

import java.text.SimpleDateFormat;

public class SimpleDateFormatBeanDefinitionParser extends AbstractSingleBeanDefinitionParser {

protected Class getBeanClass(Element element) {

return SimpleDateFormat.class;

}

protected void doParse(Element element, BeanDefinitionBuilder bean) {

// this will never be null since the schema explicitly requires that a value be supplied

String pattern = element.getAttribute("pattern");

bean.addConstructorArg(pattern);

// this however is an optional property

String lenient = element.getAttribute("lenient");

if (StringUtils.hasText(lenient)) {

bean.addPropertyValue("lenient", Boolean.valueOf(lenient));

}

}

}

We use the Spring-provided AbstractSingleBeanDefinitionParser to handle a lot of the basic grunt work of creating a single BeanDefinition.

We supply the AbstractSingleBeanDefinitionParser superclass with the type that our single BeanDefinition will represent.

In this simple case, this is all that we need to do. The creation of our single BeanDefinition is handled by the AbstractSingleBeanDefinitionParser superclass, as is the extraction and setting of the bean definition’s unique identifier.

9.2.5. Registering the handler and the schema

The coding is finished! All that remains to be done is to somehow make the Spring XML parsing infrastructure aware of our custom element; we do this by registering our custom namespaceHandler and custom XSD file in two special purpose properties files. These properties files are both placed in a 'META-INF' directory in your application, and can, for example, be distributed alongside your binary classes in a JAR file. The Spring XML parsing infrastructure will automatically pick up your new extension by consuming these special properties files, the formats of which are detailed below.

'META-INF/spring.handlers'

The properties file called 'spring.handlers' contains a mapping of XML Schema URIs to namespace handler classes. So for our example, we need to write the following:

http\://www.mycompany.com/schema/myns=org.springframework.samples.xml.MyNamespaceHandler

(The ':' character is a valid delimiter in the Java properties format, and so the ':' character in the URI needs to be escaped with a backslash.)

The first part (the key) of the key-value pair is the URI associated with your custom namespace extension, and needs to match exactly the value of the 'targetNamespace' attribute as specified in your custom XSD schema.

'META-INF/spring.schemas'

The properties file called 'spring.schemas' contains a mapping of XML Schema locations (referred to along with the schema declaration in XML files that use the schema as part of the 'xsi:schemaLocation' attribute) to classpath resources. This file is needed to prevent Spring from absolutely having to use a default EntityResolver that requires Internet access to retrieve the schema file. If you specify the mapping in this properties file, Spring will search for the schema on the classpath (in this case 'myns.xsd' in the 'org.springframework.samples.xml' package):

http\://www.mycompany.com/schema/myns/myns.xsd=org/springframework/samples/xml/myns.xsd

The upshot of this is that you are encouraged to deploy your XSD file(s) right alongside the NamespaceHandler and BeanDefinitionParser classes on the classpath.

9.2.6. Using a custom extension in your Spring XML configuration

Using a custom extension that you yourself have implemented is no different from using one of the 'custom' extensions that Spring provides straight out of the box. Find below an example of using the custom <dateformat/> element developed in the previous steps in a Spring XML configuration file.

<?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:myns="http://www.mycompany.com/schema/myns"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.mycompany.com/schema/myns http://www.mycompany.com/schema/myns/myns.xsd">

<!-- as a top-level bean -->

<myns:dateformat id="defaultDateFormat" pattern="yyyy-MM-dd HH:mm" lenient="true"/>

<bean id="jobDetailTemplate" abstract="true">

<property name="dateFormat">

<!-- as an inner bean -->

<myns:dateformat pattern="HH:mm MM-dd-yyyy"/>

</property>

</bean>

</beans>

9.2.7. Meatier examples

Find below some much meatier examples of custom XML extensions.

Nesting custom tags within custom tags

This example illustrates how you might go about writing the various artifacts required to satisfy a target of the following configuration:

<?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:foo="http://www.foo.com/schema/component"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.foo.com/schema/component http://www.foo.com/schema/component/component.xsd">

<foo:component id="bionic-family" name="Bionic-1">

<foo:component name="Mother-1">

<foo:component name="Karate-1"/>

<foo:component name="Sport-1"/>

</foo:component>

<foo:component name="Rock-1"/>

</foo:component>

</beans>

The above configuration actually nests custom extensions within each other. The class that is actually configured by the above <foo:component/> element is the Component class (shown directly below). Notice how the Component class does not expose a setter method for the 'components' property; this makes it hard (or rather impossible) to configure a bean definition for the Component class using setter injection.

package com.foo;

import java.util.ArrayList;

import java.util.List;

public class Component {

private String name;

private List<Component> components = new ArrayList<Component> ();

// mmm, there is no setter method for the 'components'

public void addComponent(Component component) {

this.components.add(component);

}

public List<Component> getComponents() {

return components;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

The typical solution to this issue is to create a custom FactoryBean that exposes a setter property for the 'components' property.

package com.foo;

import org.springframework.beans.factory.FactoryBean;

import java.util.List;

public class ComponentFactoryBean implements FactoryBean<Component> {

private Component parent;

private List<Component> children;

public void setParent(Component parent) {

this.parent = parent;

}

public void setChildren(List<Component> children) {

this.children = children;

}

public Component getObject() throws Exception {

if (this.children != null && this.children.size() > 0) {

for (Component child : children) {

this.parent.addComponent(child);

}

}

return this.parent;

}

public Class<Component> getObjectType() {

return Component.class;

}

public boolean isSingleton() {

return true;

}

}

This is all very well, and does work nicely, but exposes a lot of Spring plumbing to the end user. What we are going to do is write a custom extension that hides away all of this Spring plumbing. If we stick to the steps described previously, we’ll start off by creating the XSD schema to define the structure of our custom tag.

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/component"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://www.foo.com/schema/component"

elementFormDefault="qualified"

attributeFormDefault="unqualified">

<xsd:element name="component">

<xsd:complexType>

<xsd:choice minOccurs="0" maxOccurs="unbounded">

<xsd:element ref="component"/>

</xsd:choice>

<xsd:attribute name="id" type="xsd:ID"/>

<xsd:attribute name="name" use="required" type="xsd:string"/>

</xsd:complexType>

</xsd:element>

</xsd:schema>

We’ll then create a custom NamespaceHandler.

package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class ComponentNamespaceHandler extends NamespaceHandlerSupport {

public void init() {

registerBeanDefinitionParser("component", new ComponentBeanDefinitionParser());

}

}

Next up is the custom BeanDefinitionParser. Remember that what we are creating is a BeanDefinition describing a ComponentFactoryBean.

package com.foo;

import org.springframework.beans.factory.config.BeanDefinition;

import org.springframework.beans.factory.support.AbstractBeanDefinition;

import org.springframework.beans.factory.support.BeanDefinitionBuilder;

import org.springframework.beans.factory.support.ManagedList;

import org.springframework.beans.factory.xml.AbstractBeanDefinitionParser;

import org.springframework.beans.factory.xml.ParserContext;

import org.springframework.util.xml.DomUtils;

import org.w3c.dom.Element;

import java.util.List;

public class ComponentBeanDefinitionParser extends AbstractBeanDefinitionParser {

protected AbstractBeanDefinition parseInternal(Element element, ParserContext parserContext) {

return parseComponentElement(element);

}

private static AbstractBeanDefinition parseComponentElement(Element element) {

BeanDefinitionBuilder factory = BeanDefinitionBuilder.rootBeanDefinition(ComponentFactoryBean.class);

factory.addPropertyValue("parent", parseComponent(element));

List<Element> childElements = DomUtils.getChildElementsByTagName(element, "component");

if (childElements != null && childElements.size() > 0) {

parseChildComponents(childElements, factory);

}

return factory.getBeanDefinition();

}

private static BeanDefinition parseComponent(Element element) {

BeanDefinitionBuilder component = BeanDefinitionBuilder.rootBeanDefinition(Component.class);

component.addPropertyValue("name", element.getAttribute("name"));

return component.getBeanDefinition();

}

private static void parseChildComponents(List<Element> childElements, BeanDefinitionBuilder factory) {

ManagedList<BeanDefinition> children = new ManagedList<BeanDefinition>(childElements.size());

for (Element element : childElements) {

children.add(parseComponentElement(element));

}

factory.addPropertyValue("children", children);

}

}

Lastly, the various artifacts need to be registered with the Spring XML infrastructure.

# in 'META-INF/spring.handlers'

http\://www.foo.com/schema/component=com.foo.ComponentNamespaceHandler

# in 'META-INF/spring.schemas'

http\://www.foo.com/schema/component/component.xsd=com/foo/component.xsd

Custom attributes on 'normal' elements

Writing your own custom parser and the associated artifacts isn’t hard, but sometimes it is not the right thing to do. Consider the scenario where you need to add metadata to already existing bean definitions. In this case you certainly don’t want to have to go off and write your own entire custom extension; rather you just want to add an additional attribute to the existing bean definition element.

By way of another example, let’s say that the service class that you are defining a bean definition for a service object that will (unknown to it) be accessing a clustered JCache, and you want to ensure that the named JCache instance is eagerly started within the surrounding cluster:

<bean id="checkingAccountService" class="com.foo.DefaultCheckingAccountService"

jcache:cache-name="checking.account">

<!-- other dependencies here... -->

</bean>

What we are going to do here is create another BeanDefinition when the 'jcache:cache-name' attribute is parsed; this BeanDefinition will then initialize the named JCache for us. We will also modify the existing BeanDefinition for the 'checkingAccountService' so that it will have a dependency on this new JCache-initializing BeanDefinition.

package com.foo;

public class JCacheInitializer {

private String name;

public JCacheInitializer(String name) {

this.name = name;

}

public void initialize() {

// lots of JCache API calls to initialize the named cache...

}

}

Now onto the custom extension. Firstly, the authoring of the XSD schema describing the custom attribute (quite easy in this case).

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/jcache"

xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://www.foo.com/schema/jcache"

elementFormDefault="qualified">

<xsd:attribute name="cache-name" type="xsd:string"/>

</xsd:schema>

Next, the associated NamespaceHandler.

package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class JCacheNamespaceHandler extends NamespaceHandlerSupport {

public void init() {

super.registerBeanDefinitionDecoratorForAttribute("cache-name",

new JCacheInitializingBeanDefinitionDecorator());

}

}

Next, the parser. Note that in this case, because we are going to be parsing an XML attribute, we write a BeanDefinitionDecorator rather than a BeanDefinitionParser.

package com.foo;

import org.springframework.beans.factory.config.BeanDefinitionHolder;

import org.springframework.beans.factory.support.AbstractBeanDefinition;

import org.springframework.beans.factory.support.BeanDefinitionBuilder;

import org.springframework.beans.factory.xml.BeanDefinitionDecorator;

import org.springframework.beans.factory.xml.ParserContext;

import org.w3c.dom.Attr;

import org.w3c.dom.Node;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

public class JCacheInitializingBeanDefinitionDecorator implements BeanDefinitionDecorator {

private static final String[] EMPTY\_STRING\_ARRAY = new String[0];

public BeanDefinitionHolder decorate(Node source, BeanDefinitionHolder holder,

ParserContext ctx) {

String initializerBeanName = registerJCacheInitializer(source, ctx);

createDependencyOnJCacheInitializer(holder, initializerBeanName);

return holder;

}

private void createDependencyOnJCacheInitializer(BeanDefinitionHolder holder,

String initializerBeanName) {

AbstractBeanDefinition definition = ((AbstractBeanDefinition) holder.getBeanDefinition());

String[] dependsOn = definition.getDependsOn();

if (dependsOn == null) {

dependsOn = new String[]{initializerBeanName};

} else {

List dependencies = new ArrayList(Arrays.asList(dependsOn));

dependencies.add(initializerBeanName);

dependsOn = (String[]) dependencies.toArray(EMPTY\_STRING\_ARRAY);

}

definition.setDependsOn(dependsOn);

}

private String registerJCacheInitializer(Node source, ParserContext ctx) {

String cacheName = ((Attr) source).getValue();

String beanName = cacheName + "-initializer";

if (!ctx.getRegistry().containsBeanDefinition(beanName)) {

BeanDefinitionBuilder initializer = BeanDefinitionBuilder.rootBeanDefinition(JCacheInitializer.class);

initializer.addConstructorArg(cacheName);

ctx.getRegistry().registerBeanDefinition(beanName, initializer.getBeanDefinition());

}

return beanName;

}

}

Lastly, the various artifacts need to be registered with the Spring XML infrastructure.

# in 'META-INF/spring.handlers'

http\://www.foo.com/schema/jcache=com.foo.JCacheNamespaceHandler

# in 'META-INF/spring.schemas'

http\://www.foo.com/schema/jcache/jcache.xsd=com/foo/jcache.xsd

1. See Inversion of Control

2. See Dependency Injection

3. But see also FileSystemResource caveats.

## Testing

Version 5.0.8.RELEASE

The adoption of the test-driven-development (TDD) approach to software development is certainly advocated by the Spring team, and so coverage of Spring’s support for integration testing is covered (alongside best practices for unit testing). The Spring team has found that the correct use of IoC certainly does make both unit and integration testing easier (in that the presence of setter methods and appropriate constructors on classes makes them easier to wire together in a test without having to set up service locator registries and suchlike)…​ the chapter dedicated solely to testing will hopefully convince you of this as well.

1. Introduction to Spring Testing

Testing is an integral part of enterprise software development. This chapter focuses on the value-add of the IoC principle to unit testing and on the benefits of the Spring Framework’s support for integration testing. (A thorough treatment of testing in the enterprise is beyond the scope of this reference manual.)

2. Unit Testing

Dependency Injection should make your code less dependent on the container than it would be with traditional Java EE development. The POJOs that make up your application should be testable in JUnit or TestNG tests, with objects simply instantiated using the new operator, without Spring or any other container. You can use mock objects (in conjunction with other valuable testing techniques) to test your code in isolation. If you follow the architecture recommendations for Spring, the resulting clean layering and componentization of your codebase will facilitate easier unit testing. For example, you can test service layer objects by stubbing or mocking DAO or Repository interfaces, without needing to access persistent data while running unit tests.

True unit tests typically run extremely quickly, as there is no runtime infrastructure to set up. Emphasizing true unit tests as part of your development methodology will boost your productivity. You may not need this section of the testing chapter to help you write effective unit tests for your IoC-based applications. For certain unit testing scenarios, however, the Spring Framework provides the following mock objects and testing support classes.

2.1. Mock Objects

2.1.1. Environment

The org.springframework.mock.env package contains mock implementations of the Environment and PropertySource abstractions (see Bean definition profiles and PropertySource abstraction). MockEnvironment and MockPropertySource are useful for developing out-of-container tests for code that depends on environment-specific properties.

2.1.2. JNDI

The org.springframework.mock.jndi package contains an implementation of the JNDI SPI, which you can use to set up a simple JNDI environment for test suites or stand-alone applications. If, for example, JDBC DataSources get bound to the same JNDI names in test code as within a Java EE container, you can reuse both application code and configuration in testing scenarios without modification.

2.1.3. Servlet API

The org.springframework.mock.web package contains a comprehensive set of Servlet API mock objects that are useful for testing web contexts, controllers, and filters. These mock objects are targeted at usage with Spring’s Web MVC framework and are generally more convenient to use than dynamic mock objects such as EasyMock or alternative Servlet API mock objects such as MockObjects.

Since Spring Framework 5.0, the mock objects in org.springframework.mock.web are based on the Servlet 4.0 API.

The Spring MVC Test framework builds on the mock Servlet API objects to provide an integration testing framework for Spring MVC. See Spring MVC Test.

2.1.4. Spring Web Reactive

The package org.springframework.mock.http.server.reactive contains mock implementations of ServerHttpRequest and ServerHttpResponse for use in WebFlux applications. The package org.springframework.mock.web.server contains a mock ServerWebExchange that depends on those mock request and response objects.

Both MockServerHttpRequest and MockServerHttpResponse extend from the same abstract base classes as server-specific implementations do and share behavior with them. For example a mock request is immutable once created but you can use the mutate() method from ServerHttpRequest to create a modified instance.

In order for the mock response to properly implement the write contract and return a write completion handle (i.e. Mono<Void>), by default it uses a Flux with cache().then(), which buffers the data and makes it available for assertions in tests. Applications can set a custom write function for example to test an infinite stream.

The WebTestClient builds on the mock request and response to provide support for testing WebFlux applications without an HTTP server. The client can also be used for end-to-end tests with a running server.

2.2. Unit Testing support Classes

2.2.1. General testing utilities

The org.springframework.test.util package contains several general purpose utilities for use in unit and integration testing.

ReflectionTestUtils is a collection of reflection-based utility methods. Developers use these methods in testing scenarios where they need to change the value of a constant, set a non-public field, invoke a non-public setter method, or invoke a non-public configuration or lifecycle callback method when testing application code involving use cases such as the following.

ORM frameworks such as JPA and Hibernate that condone private or protected field access as opposed to public setter methods for properties in a domain entity.

Spring’s support for annotations such as @Autowired, @Inject, and @Resource, which provides dependency injection for private or protected fields, setter methods, and configuration methods.

Use of annotations such as @PostConstruct and @PreDestroy for lifecycle callback methods.

AopTestUtils is a collection of AOP-related utility methods. These methods can be used to obtain a reference to the underlying target object hidden behind one or more Spring proxies. For example, if you have configured a bean as a dynamic mock using a library like EasyMock or Mockito and the mock is wrapped in a Spring proxy, you may need direct access to the underlying mock in order to configure expectations on it and perform verifications. For Spring’s core AOP utilities, see AopUtils and AopProxyUtils.

2.2.2. Spring MVC

The org.springframework.test.web package contains ModelAndViewAssert, which you can use in combination with JUnit, TestNG, or any other testing framework for unit tests dealing with Spring MVC ModelAndView objects.

Unit testing Spring MVC Controllers

To unit test your Spring MVC Controllers as POJOs, use ModelAndViewAssert combined with MockHttpServletRequest, MockHttpSession, and so on from Spring’s Servlet API mocks. For thorough integration testing of your Spring MVC and REST Controllers in conjunction with your WebApplicationContext configuration for Spring MVC, use the Spring MVC Test Framework instead.

3. Integration Testing

3.1. Overview

It is important to be able to perform some integration testing without requiring deployment to your application server or connecting to other enterprise infrastructure. This will enable you to test things such as:

The correct wiring of your Spring IoC container contexts.

Data access using JDBC or an ORM tool. This would include such things as the correctness of SQL statements, Hibernate queries, JPA entity mappings, etc.

The Spring Framework provides first-class support for integration testing in the spring-test module. The name of the actual JAR file might include the release version and might also be in the long org.springframework.test form, depending on where you get it from (see the section on Dependency Management for an explanation). This library includes the org.springframework.test package, which contains valuable classes for integration testing with a Spring container. This testing does not rely on an application server or other deployment environment. Such tests are slower to run than unit tests but much faster than the equivalent Selenium tests or remote tests that rely on deployment to an application server.

In Spring 2.5 and later, unit and integration testing support is provided in the form of the annotation-driven Spring TestContext Framework. The TestContext framework is agnostic of the actual testing framework in use, thus allowing instrumentation of tests in various environments including JUnit, TestNG, and so on.

3.2. Goals of Integration Testing

Spring’s integration testing support has the following primary goals:

To manage Spring IoC container caching between test execution.

To provide Dependency Injection of test fixture instances.

To provide transaction management appropriate to integration testing.

To supply Spring-specific base classes that assist developers in writing integration tests.

The next few sections describe each goal and provide links to implementation and configuration details.

3.2.1. Context management and caching

The Spring TestContext Framework provides consistent loading of Spring ApplicationContexts and WebApplicationContexts as well as caching of those contexts. Support for the caching of loaded contexts is important, because startup time can become an issue — not because of the overhead of Spring itself, but because the objects instantiated by the Spring container take time to instantiate. For example, a project with 50 to 100 Hibernate mapping files might take 10 to 20 seconds to load the mapping files, and incurring that cost before running every test in every test fixture leads to slower overall test runs that reduce developer productivity.

Test classes typically declare either an array of resource locations for XML or Groovy configuration metadata — often in the classpath — or an array of annotated classes that is used to configure the application. These locations or classes are the same as or similar to those specified in web.xml or other configuration files for production deployments.

By default, once loaded, the configured ApplicationContext is reused for each test. Thus the setup cost is incurred only once per test suite, and subsequent test execution is much faster. In this context, the term test suite means all tests run in the same JVM — for example, all tests run from an Ant, Maven, or Gradle build for a given project or module. In the unlikely case that a test corrupts the application context and requires reloading — for example, by modifying a bean definition or the state of an application object — the TestContext framework can be configured to reload the configuration and rebuild the application context before executing the next test.

See Context management and Context caching with the TestContext framework.

3.2.2. Dependency Injection of test fixtures

When the TestContext framework loads your application context, it can optionally configure instances of your test classes via Dependency Injection. This provides a convenient mechanism for setting up test fixtures using preconfigured beans from your application context. A strong benefit here is that you can reuse application contexts across various testing scenarios (e.g., for configuring Spring-managed object graphs, transactional proxies, DataSources, etc.), thus avoiding the need to duplicate complex test fixture setup for individual test cases.

As an example, consider the scenario where we have a class, HibernateTitleRepository, that implements data access logic for a Title domain entity. We want to write integration tests that test the following areas:

The Spring configuration: basically, is everything related to the configuration of the HibernateTitleRepository bean correct and present?

The Hibernate mapping file configuration: is everything mapped correctly, and are the correct lazy-loading settings in place?

The logic of the HibernateTitleRepository: does the configured instance of this class perform as anticipated?

See dependency injection of test fixtures with the TestContext framework.

3.2.3. Transaction management

One common issue in tests that access a real database is their effect on the state of the persistence store. Even when you’re using a development database, changes to the state may affect future tests. Also, many operations — such as inserting or modifying persistent data — cannot be performed (or verified) outside a transaction.

The TestContext framework addresses this issue. By default, the framework will create and roll back a transaction for each test. You simply write code that can assume the existence of a transaction. If you call transactionally proxied objects in your tests, they will behave correctly, according to their configured transactional semantics. In addition, if a test method deletes the contents of selected tables while running within the transaction managed for the test, the transaction will roll back by default, and the database will return to its state prior to execution of the test. Transactional support is provided to a test via a PlatformTransactionManager bean defined in the test’s application context.

If you want a transaction to commit — unusual, but occasionally useful when you want a particular test to populate or modify the database — the TestContext framework can be instructed to cause the transaction to commit instead of roll back via the @Commit annotation.

See transaction management with the TestContext framework.

3.2.4. Support classes for integration testing

The Spring TestContext Framework provides several abstract support classes that simplify the writing of integration tests. These base test classes provide well-defined hooks into the testing framework as well as convenient instance variables and methods, which enable you to access:

The ApplicationContext, for performing explicit bean lookups or testing the state of the context as a whole.

A JdbcTemplate, for executing SQL statements to query the database. Such queries can be used to confirm database state both prior to and after execution of database-related application code, and Spring ensures that such queries run in the scope of the same transaction as the application code. When used in conjunction with an ORM tool, be sure to avoid false positives.

In addition, you may want to create your own custom, application-wide superclass with instance variables and methods specific to your project.

See support classes for the TestContext framework.

3.3. JDBC Testing Support

The org.springframework.test.jdbc package contains JdbcTestUtils, which is a collection of JDBC related utility functions intended to simplify standard database testing scenarios. Specifically, JdbcTestUtils provides the following static utility methods.

countRowsInTable(..): counts the number of rows in the given table

countRowsInTableWhere(..): counts the number of rows in the given table, using the provided WHERE clause

deleteFromTables(..): deletes all rows from the specified tables

deleteFromTableWhere(..): deletes rows from the given table, using the provided WHERE clause

dropTables(..): drops the specified tables

Note that AbstractTransactionalJUnit4SpringContextTests and AbstractTransactionalTestNGSpringContextTests provide convenience methods which delegate to the aforementioned methods in JdbcTestUtils.

The spring-jdbc module provides support for configuring and launching an embedded database which can be used in integration tests that interact with a database. For details, see Embedded database support and Testing data access logic with an embedded database.

3.4. Annotations

3.4.1. Spring Testing Annotations

The Spring Framework provides the following set of Spring-specific annotations that you can use in your unit and integration tests in conjunction with the TestContext framework. Refer to the corresponding javadocs for further information, including default attribute values, attribute aliases, and so on.

@BootstrapWith

@BootstrapWith is a class-level annotation that is used to configure how the Spring TestContext Framework is bootstrapped. Specifically, @BootstrapWith is used to specify a custom TestContextBootstrapper. Consult the Bootstrapping the TestContext framework section for further details.

@ContextConfiguration

@ContextConfiguration defines class-level metadata that is used to determine how to load and configure an ApplicationContext for integration tests. Specifically, @ContextConfiguration declares the application context resource locations or the annotated classes that will be used to load the context.

Resource locations are typically XML configuration files or Groovy scripts located in the classpath; whereas, annotated classes are typically @Configuration classes. However, resource locations can also refer to files and scripts in the file system, and annotated classes can be component classes, etc.

@ContextConfiguration("/test-config.xml")

public class XmlApplicationContextTests {

// class body...

}

@ContextConfiguration(classes = TestConfig.class)

public class ConfigClassApplicationContextTests {

// class body...

}

As an alternative or in addition to declaring resource locations or annotated classes, @ContextConfiguration may be used to declare ApplicationContextInitializer classes.

@ContextConfiguration(initializers = CustomContextIntializer.class)

public class ContextInitializerTests {

// class body...

}

@ContextConfiguration may optionally be used to declare the ContextLoader strategy as well. Note, however, that you typically do not need to explicitly configure the loader since the default loader supports either resource locations or annotated classes as well as initializers.

@ContextConfiguration(locations = "/test-context.xml", loader = CustomContextLoader.class)

public class CustomLoaderXmlApplicationContextTests {

// class body...

}

@ContextConfiguration provides support for inheriting resource locations or configuration classes as well as context initializers declared by superclasses by default.

See Context management and the @ContextConfiguration javadocs for further details.

@WebAppConfiguration

@WebAppConfiguration is a class-level annotation that is used to declare that the ApplicationContext loaded for an integration test should be a WebApplicationContext. The mere presence of @WebAppConfiguration on a test class ensures that a WebApplicationContext will be loaded for the test, using the default value of "file:src/main/webapp" for the path to the root of the web application (i.e., the resource base path). The resource base path is used behind the scenes to create a MockServletContext which serves as the ServletContext for the test’s WebApplicationContext.

@ContextConfiguration

@WebAppConfiguration

public class WebAppTests {

// class body...

}

To override the default, specify a different base resource path via the implicit value attribute. Both classpath: and file: resource prefixes are supported. If no resource prefix is supplied the path is assumed to be a file system resource.

@ContextConfiguration

@WebAppConfiguration("classpath:test-web-resources")

public class WebAppTests {

// class body...

}

Note that @WebAppConfiguration must be used in conjunction with @ContextConfiguration, either within a single test class or within a test class hierarchy. See the @WebAppConfiguration javadocs for further details.

@ContextHierarchy

@ContextHierarchy is a class-level annotation that is used to define a hierarchy of ApplicationContexts for integration tests. @ContextHierarchy should be declared with a list of one or more @ContextConfiguration instances, each of which defines a level in the context hierarchy. The following examples demonstrate the use of @ContextHierarchy within a single test class; however, @ContextHierarchy can also be used within a test class hierarchy.

@ContextHierarchy({

@ContextConfiguration("/parent-config.xml"),

@ContextConfiguration("/child-config.xml")

})

public class ContextHierarchyTests {

// class body...

}

@WebAppConfiguration

@ContextHierarchy({

@ContextConfiguration(classes = AppConfig.class),

@ContextConfiguration(classes = WebConfig.class)

})

public class WebIntegrationTests {

// class body...

}

If you need to merge or override the configuration for a given level of the context hierarchy within a test class hierarchy, you must explicitly name that level by supplying the same value to the name attribute in @ContextConfiguration at each corresponding level in the class hierarchy. See Context hierarchies and the @ContextHierarchy javadocs for further examples.

@ActiveProfiles

@ActiveProfiles is a class-level annotation that is used to declare which bean definition profiles should be active when loading an ApplicationContext for an integration test.

@ContextConfiguration

@ActiveProfiles("dev")

public class DeveloperTests {

// class body...

}

@ContextConfiguration

@ActiveProfiles({"dev", "integration"})

public class DeveloperIntegrationTests {

// class body...

}

@ActiveProfiles provides support for inheriting active bean definition profiles declared by superclasses by default. It is also possible to resolve active bean definition profiles programmatically by implementing a custom ActiveProfilesResolver and registering it via the resolver attribute of @ActiveProfiles.

See Context configuration with environment profiles and the @ActiveProfiles javadocs for examples and further details.

@TestPropertySource

@TestPropertySource is a class-level annotation that is used to configure the locations of properties files and inlined properties to be added to the set of PropertySources in the Environment for an ApplicationContext loaded for an integration test.

Test property sources have higher precedence than those loaded from the operating system’s environment or Java system properties as well as property sources added by the application declaratively via @PropertySource or programmatically. Thus, test property sources can be used to selectively override properties defined in system and application property sources. Furthermore, inlined properties have higher precedence than properties loaded from resource locations.

The following example demonstrates how to declare a properties file from the classpath.

@ContextConfiguration

@TestPropertySource("/test.properties")

public class MyIntegrationTests {

// class body...

}

The following example demonstrates how to declare inlined properties.

@ContextConfiguration

@TestPropertySource(properties = { "timezone = GMT", "port: 4242" })

public class MyIntegrationTests {

// class body...

}

@DirtiesContext

@DirtiesContext indicates that the underlying Spring ApplicationContext has been dirtied during the execution of a test (i.e., modified or corrupted in some manner — for example, by changing the state of a singleton bean) and should be closed. When an application context is marked dirty, it is removed from the testing framework’s cache and closed. As a consequence, the underlying Spring container will be rebuilt for any subsequent test that requires a context with the same configuration metadata.

@DirtiesContext can be used as both a class-level and method-level annotation within the same class or class hierarchy. In such scenarios, the ApplicationContext is marked as dirty before or after any such annotated method as well as before or after the current test class, depending on the configured methodMode and classMode.

The following examples explain when the context would be dirtied for various configuration scenarios:

Before the current test class, when declared on a class with class mode set to BEFORE\_CLASS.

@DirtiesContext(classMode = BEFORE\_CLASS)

public class FreshContextTests {

// some tests that require a new Spring container

}

After the current test class, when declared on a class with class mode set to AFTER\_CLASS (i.e., the default class mode).

@DirtiesContext

public class ContextDirtyingTests {

// some tests that result in the Spring container being dirtied

}

Before each test method in the current test class, when declared on a class with class mode set to BEFORE\_EACH\_TEST\_METHOD.

@DirtiesContext(classMode = BEFORE\_EACH\_TEST\_METHOD)

public class FreshContextTests {

// some tests that require a new Spring container

}

After each test method in the current test class, when declared on a class with class mode set to AFTER\_EACH\_TEST\_METHOD.

@DirtiesContext(classMode = AFTER\_EACH\_TEST\_METHOD)

public class ContextDirtyingTests {

// some tests that result in the Spring container being dirtied

}

Before the current test, when declared on a method with the method mode set to BEFORE\_METHOD.

@DirtiesContext(methodMode = BEFORE\_METHOD)

@Test

public void testProcessWhichRequiresFreshAppCtx() {

// some logic that requires a new Spring container

}

After the current test, when declared on a method with the method mode set to AFTER\_METHOD (i.e., the default method mode).

@DirtiesContext

@Test

public void testProcessWhichDirtiesAppCtx() {

// some logic that results in the Spring container being dirtied

}

If @DirtiesContext is used in a test whose context is configured as part of a context hierarchy via @ContextHierarchy, the hierarchyMode flag can be used to control how the context cache is cleared. By default an exhaustive algorithm will be used that clears the context cache including not only the current level but also all other context hierarchies that share an ancestor context common to the current test; all ApplicationContexts that reside in a sub-hierarchy of the common ancestor context will be removed from the context cache and closed. If the exhaustive algorithm is overkill for a particular use case, the simpler current level algorithm can be specified instead, as seen below.

@ContextHierarchy({

@ContextConfiguration("/parent-config.xml"),

@ContextConfiguration("/child-config.xml")

})

public class BaseTests {

// class body...

}

public class ExtendedTests extends BaseTests {

@Test

@DirtiesContext(hierarchyMode = CURRENT\_LEVEL)

public void test() {

// some logic that results in the child context being dirtied

}

}

For further details regarding the EXHAUSTIVE and CURRENT\_LEVEL algorithms see the DirtiesContext.HierarchyMode javadocs.

@TestExecutionListeners

@TestExecutionListeners defines class-level metadata for configuring the TestExecutionListener implementations that should be registered with the TestContextManager. Typically, @TestExecutionListeners is used in conjunction with @ContextConfiguration.

@ContextConfiguration

@TestExecutionListeners({CustomTestExecutionListener.class, AnotherTestExecutionListener.class})

public class CustomTestExecutionListenerTests {

// class body...

}

@TestExecutionListeners supports inherited listeners by default. See the javadocs for an example and further details.

@Commit

@Commit indicates that the transaction for a transactional test method should be committed after the test method has completed. @Commit can be used as a direct replacement for @Rollback(false) in order to more explicitly convey the intent of the code. Analogous to @Rollback, @Commit may also be declared as a class-level or method-level annotation.

@Commit

@Test

public void testProcessWithoutRollback() {

// ...

}

@Rollback

@Rollback indicates whether the transaction for a transactional test method should be rolled back after the test method has completed. If true, the transaction is rolled back; otherwise, the transaction is committed (see also @Commit). Rollback semantics for integration tests in the Spring TestContext Framework default to true even if @Rollback is not explicitly declared.

When declared as a class-level annotation, @Rollback defines the default rollback semantics for all test methods within the test class hierarchy. When declared as a method-level annotation, @Rollback defines rollback semantics for the specific test method, potentially overriding class-level @Rollback or @Commit semantics.

@Rollback(false)

@Test

public void testProcessWithoutRollback() {

// ...

}

@BeforeTransaction

@BeforeTransaction indicates that the annotated void method should be executed before a transaction is started for test methods configured to run within a transaction via Spring’s @Transactional annotation. As of Spring Framework 4.3, @BeforeTransaction methods are not required to be public and may be declared on Java 8 based interface default methods.

@BeforeTransaction

void beforeTransaction() {

// logic to be executed before a transaction is started

}

@AfterTransaction

@AfterTransaction indicates that the annotated void method should be executed after a transaction is ended for test methods configured to run within a transaction via Spring’s @Transactional annotation. As of Spring Framework 4.3, @AfterTransaction methods are not required to be public and may be declared on Java 8 based interface default methods.

@AfterTransaction

void afterTransaction() {

// logic to be executed after a transaction has ended

}

@Sql

@Sql is used to annotate a test class or test method to configure SQL scripts to be executed against a given database during integration tests.

@Test

@Sql({"/test-schema.sql", "/test-user-data.sql"})

public void userTest {

// execute code that relies on the test schema and test data

}

See Executing SQL scripts declaratively with @Sql for further details.

@SqlConfig

@SqlConfig defines metadata that is used to determine how to parse and execute SQL scripts configured via the @Sql annotation.

@Test

@Sql(

scripts = "/test-user-data.sql",

config = @SqlConfig(commentPrefix = "`", separator = "@@")

)

public void userTest {

// execute code that relies on the test data

}

@SqlGroup

@SqlGroup is a container annotation that aggregates several @Sql annotations. @SqlGroup can be used natively, declaring several nested @Sql annotations, or it can be used in conjunction with Java 8’s support for repeatable annotations, where @Sql can simply be declared several times on the same class or method, implicitly generating this container annotation.

@Test

@SqlGroup({

@Sql(scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "`")),

@Sql("/test-user-data.sql")

)}

public void userTest {

// execute code that uses the test schema and test data

}

3.4.2. Standard Annotation Support

The following annotations are supported with standard semantics for all configurations of the Spring TestContext Framework. Note that these annotations are not specific to tests and can be used anywhere in the Spring Framework.

@Autowired

@Qualifier

@Resource (javax.annotation) if JSR-250 is present

@ManagedBean (javax.annotation) if JSR-250 is present

@Inject (javax.inject) if JSR-330 is present

@Named (javax.inject) if JSR-330 is present

@PersistenceContext (javax.persistence) if JPA is present

@PersistenceUnit (javax.persistence) if JPA is present

@Required

@Transactional

JSR-250 Lifecycle Annotations

In the Spring TestContext Framework @PostConstruct and @PreDestroy may be used with standard semantics on any application components configured in the ApplicationContext; however, these lifecycle annotations have limited usage within an actual test class.

If a method within a test class is annotated with @PostConstruct, that method will be executed before any before methods of the underlying test framework (e.g., methods annotated with JUnit Jupiter’s @BeforeEach), and that will apply for every test method in the test class. On the other hand, if a method within a test class is annotated with @PreDestroy, that method will never be executed. Within a test class it is therefore recommended to use test lifecycle callbacks from the underlying test framework instead of @PostConstruct and @PreDestroy.

3.4.3. Spring JUnit 4 Testing Annotations

The following annotations are only supported when used in conjunction with the SpringRunner, Spring’s JUnit 4 rules, or Spring’s JUnit 4 support classes.

@IfProfileValue

@IfProfileValue indicates that the annotated test is enabled for a specific testing environment. If the configured ProfileValueSource returns a matching value for the provided name, the test is enabled. Otherwise, the test will be disabled and effectively ignored.

@IfProfileValue can be applied at the class level, the method level, or both. Class-level usage of @IfProfileValue takes precedence over method-level usage for any methods within that class or its subclasses. Specifically, a test is enabled if it is enabled both at the class level and at the method level; the absence of @IfProfileValue means the test is implicitly enabled. This is analogous to the semantics of JUnit 4’s @Ignore annotation, except that the presence of @Ignore always disables a test.

@IfProfileValue(name="java.vendor", value="Oracle Corporation")

@Test

public void testProcessWhichRunsOnlyOnOracleJvm() {

// some logic that should run only on Java VMs from Oracle Corporation

}

Alternatively, you can configure @IfProfileValue with a list of values (with OR semantics) to achieve TestNG-like support for test groups in a JUnit 4 environment. Consider the following example:

@IfProfileValue(name="test-groups", values={"unit-tests", "integration-tests"})

@Test

public void testProcessWhichRunsForUnitOrIntegrationTestGroups() {

// some logic that should run only for unit and integration test groups

}

@ProfileValueSourceConfiguration

@ProfileValueSourceConfiguration is a class-level annotation that specifies what type of ProfileValueSource to use when retrieving profile values configured through the @IfProfileValue annotation. If @ProfileValueSourceConfiguration is not declared for a test, SystemProfileValueSource is used by default.

@ProfileValueSourceConfiguration(CustomProfileValueSource.class)

public class CustomProfileValueSourceTests {

// class body...

}

@Timed

@Timed indicates that the annotated test method must finish execution in a specified time period (in milliseconds). If the text execution time exceeds the specified time period, the test fails.

The time period includes execution of the test method itself, any repetitions of the test (see @Repeat), as well as any set up or tear down of the test fixture.

@Timed(millis=1000)

public void testProcessWithOneSecondTimeout() {

// some logic that should not take longer than 1 second to execute

}

Spring’s @Timed annotation has different semantics than JUnit 4’s @Test(timeout=…​) support. Specifically, due to the manner in which JUnit 4 handles test execution timeouts (that is, by executing the test method in a separate Thread), @Test(timeout=…​) preemptively fails the test if the test takes too long. Spring’s @Timed, on the other hand, does not preemptively fail the test but rather waits for the test to complete before failing.

@Repeat

@Repeat indicates that the annotated test method must be executed repeatedly. The number of times that the test method is to be executed is specified in the annotation.

The scope of execution to be repeated includes execution of the test method itself as well as any set up or tear down of the test fixture.

@Repeat(10)

@Test

public void testProcessRepeatedly() {

// ...

}

3.4.4. Spring JUnit Jupiter Testing Annotations

The following annotations are only supported when used in conjunction with the SpringExtension and JUnit Jupiter (i.e., the programming model in JUnit 5).

@SpringJUnitConfig

@SpringJUnitConfig is a composed annotation that combines @ExtendWith(SpringExtension.class) from JUnit Jupiter with @ContextConfiguration from the Spring TestContext Framework. It can be used at the class level as a drop-in replacement for @ContextConfiguration. With regard to configuration options, the only difference between @ContextConfiguration and @SpringJUnitConfig is that annotated classes may be declared via the value attribute in @SpringJUnitConfig.

@SpringJUnitConfig(TestConfig.class)

class ConfigurationClassJUnitJupiterSpringTests {

// class body...

}

@SpringJUnitConfig(locations = "/test-config.xml")

class XmlJUnitJupiterSpringTests {

// class body...

}

See Context management as well as the javadocs for @SpringJUnitConfig and @ContextConfiguration for further details.

@SpringJUnitWebConfig

@SpringJUnitWebConfig is a composed annotation that combines @ExtendWith(SpringExtension.class) from JUnit Jupiter with @ContextConfiguration and @WebAppConfiguration from the Spring TestContext Framework. It can be used at the class level as a drop-in replacement for @ContextConfiguration and @WebAppConfiguration. With regard to configuration options, the only difference between @ContextConfiguration and @SpringJUnitWebConfig is that annotated classes may be declared via the value attribute in @SpringJUnitWebConfig. In addition, the value attribute from @WebAppConfiguration can only be overridden via the resourcePath attribute in @SpringJUnitWebConfig.

@SpringJUnitWebConfig(TestConfig.class)

class ConfigurationClassJUnitJupiterSpringWebTests {

// class body...

}

@SpringJUnitWebConfig(locations = "/test-config.xml")

class XmlJUnitJupiterSpringWebTests {

// class body...

}

See Context management as well as the javadocs for @SpringJUnitWebConfig, @ContextConfiguration, and @WebAppConfiguration for further details.

@EnabledIf

@EnabledIf is used to signal that the annotated JUnit Jupiter test class or test method is enabled and should be executed if the supplied expression evaluates to true. Specifically, if the expression evaluates to Boolean.TRUE or a String equal to "true" (ignoring case), the test will be enabled. When applied at the class level, all test methods within that class are automatically enabled by default as well.

Expressions can be any of the following.

Spring Expression Language (SpEL) expression – for example:

@EnabledIf("#{systemProperties['os.name'].toLowerCase().contains('mac')}")

Placeholder for a property available in the Spring Environment – for example:

@EnabledIf("${smoke.tests.enabled}")

Text literal – for example:

@EnabledIf("true")

Note, however, that a text literal which is not the result of dynamic resolution of a property placeholder is of zero practical value since @EnabledIf("false") is equivalent to @Disabled and @EnabledIf("true") is logically meaningless.

@EnabledIf may be used as a meta-annotation to create custom composed annotations. For example, a custom @EnabledOnMac annotation can be created as follows.

@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@EnabledIf(

expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",

reason = "Enabled on Mac OS"

)

public @interface EnabledOnMac {}

@DisabledIf

@DisabledIf is used to signal that the annotated JUnit Jupiter test class or test method is disabled and should not be executed if the supplied expression evaluates to true. Specifically, if the expression evaluates to Boolean.TRUE or a String equal to "true" (ignoring case), the test will be disabled. When applied at the class level, all test methods within that class are automatically disabled as well.

Expressions can be any of the following.

Spring Expression Language (SpEL) expression – for example:

@DisabledIf("#{systemProperties['os.name'].toLowerCase().contains('mac')}")

Placeholder for a property available in the Spring Environment – for example:

@DisabledIf("${smoke.tests.disabled}")

Text literal – for example:

@DisabledIf("true")

Note, however, that a text literal which is not the result of dynamic resolution of a property placeholder is of zero practical value since @DisabledIf("true") is equivalent to @Disabled and @DisabledIf("false") is logically meaningless.

@DisabledIf may be used as a meta-annotation to create custom composed annotations. For example, a custom @DisabledOnMac annotation can be created as follows.

@Target({ElementType.TYPE, ElementType.METHOD})

@Retention(RetentionPolicy.RUNTIME)

@DisabledIf(

expression = "#{systemProperties['os.name'].toLowerCase().contains('mac')}",

reason = "Disabled on Mac OS"

)

public @interface DisabledOnMac {}

3.4.5. Meta-Annotation Support for Testing

It is possible to use most test-related annotations as meta-annotations in order to create custom composed annotations and reduce configuration duplication across a test suite.

Each of the following may be used as meta-annotations in conjunction with the TestContext framework.

@BootstrapWith

@ContextConfiguration

@ContextHierarchy

@ActiveProfiles

@TestPropertySource

@DirtiesContext

@WebAppConfiguration

@TestExecutionListeners

@Transactional

@BeforeTransaction

@AfterTransaction

@Commit

@Rollback

@Sql

@SqlConfig

@SqlGroup

@Repeat (only supported on JUnit 4)

@Timed (only supported on JUnit 4)

@IfProfileValue (only supported on JUnit 4)

@ProfileValueSourceConfiguration (only supported on JUnit 4)

@SpringJUnitConfig (only supported on JUnit Jupiter)

@SpringJUnitWebConfig (only supported on JUnit Jupiter)

@EnabledIf (only supported on JUnit Jupiter)

@DisabledIf (only supported on JUnit Jupiter)

For example, if we discover that we are repeating the following configuration across our JUnit 4 based test suite…​

@RunWith(SpringRunner.class)

@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})

@ActiveProfiles("dev")

@Transactional

public class OrderRepositoryTests { }

@RunWith(SpringRunner.class)

@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})

@ActiveProfiles("dev")

@Transactional

public class UserRepositoryTests { }

We can reduce the above duplication by introducing a custom composed annotation that centralizes the common test configuration for Spring like this:

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})

@ActiveProfiles("dev")

@Transactional

public @interface TransactionalDevTestConfig { }

Then we can use our custom @TransactionalDevTestConfig annotation to simplify the configuration of individual JUnit 4 based test classes as follows:

@RunWith(SpringRunner.class)

@TransactionalDevTestConfig

public class OrderRepositoryTests { }

@RunWith(SpringRunner.class)

@TransactionalDevTestConfig

public class UserRepositoryTests { }

If we are writing tests using JUnit Jupiter, we can reduce code duplication even further since annotations in JUnit 5 can also be used as meta-annotations. For example, if we discover that we are repeating the following configuration across our JUnit Jupiter based test suite…​

@ExtendWith(SpringExtension.class)

@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})

@ActiveProfiles("dev")

@Transactional

class OrderRepositoryTests { }

@ExtendWith(SpringExtension.class)

@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})

@ActiveProfiles("dev")

@Transactional

class UserRepositoryTests { }

We can reduce the above duplication by introducing a custom composed annotation that centralizes the common test configuration for Spring and JUnit Jupiter like this:

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@ExtendWith(SpringExtension.class)

@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})

@ActiveProfiles("dev")

@Transactional

public @interface TransactionalDevTestConfig { }

Then we can use our custom @TransactionalDevTestConfig annotation to simplify the configuration of individual JUnit Jupiter based test classes as follows:

@TransactionalDevTestConfig

class OrderRepositoryTests { }

@TransactionalDevTestConfig

class UserRepositoryTests { }

Since JUnit Jupiter supports the use of @Test, @RepeatedTest, ParameterizedTest, etc. as meta-annotations, it is also possible to create custom composed annotations at the test method level. For example, if we wish to create a composed annotation that combines the @Test and @Tag annotations from JUnit Jupiter with the @Transactional annotation from Spring, we could create an @TransactionalIntegrationTest annotation as follows.

@Target(ElementType.METHOD)

@Retention(RetentionPolicy.RUNTIME)

@Transactional

@Tag("integration-test") // org.junit.jupiter.api.Tag

@Test // org.junit.jupiter.api.Test

public @interface TransactionalIntegrationTest { }

Then we can use our custom @TransactionalIntegrationTest annotation to simplify the configuration of individual JUnit Jupiter based test methods as follows:

@TransactionalIntegrationTest

void saveOrder() { }

@TransactionalIntegrationTest

void deleteOrder() { }

For further details, consult the Spring Annotation Programming Model.

3.5. Spring TestContext Framework

The Spring TestContext Framework (located in the org.springframework.test.context package) provides generic, annotation-driven unit and integration testing support that is agnostic of the testing framework in use. The TestContext framework also places a great deal of importance on convention over configuration with reasonable defaults that can be overridden through annotation-based configuration.

In addition to generic testing infrastructure, the TestContext framework provides explicit support for JUnit 4, JUnit Jupiter (a.k.a., JUnit 5), and TestNG. For JUnit 4 and TestNG, Spring provides abstract support classes. Furthermore, Spring provides a custom JUnit Runner and custom JUnit Rules for JUnit 4 as well as a custom Extension for JUnit Jupiter that allow one to write so-called POJO test classes. POJO test classes are not required to extend a particular class hierarchy such as the abstract support classes.

The following section provides an overview of the internals of the TestContext framework. If you are only interested in using the framework and not necessarily interested in extending it with your own custom listeners or custom loaders, feel free to go directly to the configuration (context management, dependency injection, transaction management), support classes, and annotation support sections.

3.5.1. Key abstractions

The core of the framework consists of the TestContextManager class and the TestContext, TestExecutionListener, and SmartContextLoader interfaces. A TestContextManager is created per test class (e.g., for the execution of all test methods within a single test class in JUnit Jupiter). The TestContextManager in turn manages a TestContext that holds the context of the current test. The TestContextManager also updates the state of the TestContext as the test progresses and delegates to TestExecutionListener implementations, which instrument the actual test execution by providing dependency injection, managing transactions, and so on. A SmartContextLoader is responsible for loading an ApplicationContext for a given test class. Consult the javadocs and the Spring test suite for further information and examples of various implementations.

TestContext

TestContext encapsulates the context in which a test is executed, agnostic of the actual testing framework in use, and provides context management and caching support for the test instance for which it is responsible. The TestContext also delegates to a SmartContextLoader to load an ApplicationContext if requested.

TestContextManager

TestContextManager is the main entry point into the Spring TestContext Framework and is responsible for managing a single TestContext and signaling events to each registered TestExecutionListener at well-defined test execution points:

prior to any before class or before all methods of a particular testing framework

test instance post-processing

prior to any before or before each methods of a particular testing framework

immediately before execution of the test method but after test setup

immediately after execution of the test method but before test tear down

after any after or after each methods of a particular testing framework

after any after class or after all methods of a particular testing framework

TestExecutionListener

TestExecutionListener defines the API for reacting to test execution events published by the TestContextManager with which the listener is registered. See TestExecutionListener configuration.

Context Loaders

ContextLoader is a strategy interface that was introduced in Spring 2.5 for loading an ApplicationContext for an integration test managed by the Spring TestContext Framework. Implement SmartContextLoader instead of this interface in order to provide support for annotated classes, active bean definition profiles, test property sources, context hierarchies, and WebApplicationContext support.

SmartContextLoader is an extension of the ContextLoader interface introduced in Spring 3.1. The SmartContextLoader SPI supersedes the ContextLoader SPI that was introduced in Spring 2.5. Specifically, a SmartContextLoader can choose to process resource locations, annotated classes, or context initializers. Furthermore, a SmartContextLoader can set active bean definition profiles and test property sources in the context that it loads.

Spring provides the following implementations:

DelegatingSmartContextLoader: one of two default loaders which delegates internally to an AnnotationConfigContextLoader, a GenericXmlContextLoader, or a GenericGroovyXmlContextLoader depending either on the configuration declared for the test class or on the presence of default locations or default configuration classes. Groovy support is only enabled if Groovy is on the classpath.

WebDelegatingSmartContextLoader: one of two default loaders which delegates internally to an AnnotationConfigWebContextLoader, a GenericXmlWebContextLoader, or a GenericGroovyXmlWebContextLoader depending either on the configuration declared for the test class or on the presence of default locations or default configuration classes. A web ContextLoader will only be used if @WebAppConfiguration is present on the test class. Groovy support is only enabled if Groovy is on the classpath.

AnnotationConfigContextLoader: loads a standard ApplicationContext from annotated classes.

AnnotationConfigWebContextLoader: loads a WebApplicationContext from annotated classes.

GenericGroovyXmlContextLoader: loads a standard ApplicationContext from resource locations that are either Groovy scripts or XML configuration files.

GenericGroovyXmlWebContextLoader: loads a WebApplicationContext from resource locations that are either Groovy scripts or XML configuration files.

GenericXmlContextLoader: loads a standard ApplicationContext from XML resource locations.

GenericXmlWebContextLoader: loads a WebApplicationContext from XML resource locations.

GenericPropertiesContextLoader: loads a standard ApplicationContext from Java Properties files.

3.5.2. Bootstrapping the TestContext framework

The default configuration for the internals of the Spring TestContext Framework is sufficient for all common use cases. However, there are times when a development team or third party framework would like to change the default ContextLoader, implement a custom TestContext or ContextCache, augment the default sets of ContextCustomizerFactory and TestExecutionListener implementations, etc. For such low level control over how the TestContext framework operates, Spring provides a bootstrapping strategy.

TestContextBootstrapper defines the SPI for bootstrapping the TestContext framework. A TestContextBootstrapper is used by the TestContextManager to load the TestExecutionListener implementations for the current test and to build the TestContext that it manages. A custom bootstrapping strategy can be configured for a test class (or test class hierarchy) via @BootstrapWith, either directly or as a meta-annotation. If a bootstrapper is not explicitly configured via @BootstrapWith, either the DefaultTestContextBootstrapper or the WebTestContextBootstrapper will be used, depending on the presence of @WebAppConfiguration.

Since the TestContextBootstrapper SPI is likely to change in the future in order to accommodate new requirements, implementers are strongly encouraged not to implement this interface directly but rather to extend AbstractTestContextBootstrapper or one of its concrete subclasses instead.

3.5.3. TestExecutionListener configuration

Spring provides the following TestExecutionListener implementations that are registered by default, exactly in this order.

ServletTestExecutionListener: configures Servlet API mocks for a WebApplicationContext

DirtiesContextBeforeModesTestExecutionListener: handles the @DirtiesContext annotation for before modes

DependencyInjectionTestExecutionListener: provides dependency injection for the test instance

DirtiesContextTestExecutionListener: handles the @DirtiesContext annotation for after modes

TransactionalTestExecutionListener: provides transactional test execution with default rollback semantics

SqlScriptsTestExecutionListener: executes SQL scripts configured via the @Sql annotation

Registering custom TestExecutionListeners

Custom TestExecutionListeners can be registered for a test class and its subclasses via the @TestExecutionListeners annotation. See annotation support and the javadocs for @TestExecutionListeners for details and examples.

Automatic discovery of default TestExecutionListeners

Registering custom TestExecutionListeners via @TestExecutionListeners is suitable for custom listeners that are used in limited testing scenarios; however, it can become cumbersome if a custom listener needs to be used across a test suite. Since Spring Framework 4.1, this issue is addressed via support for automatic discovery of default TestExecutionListener implementations via the SpringFactoriesLoader mechanism.

Specifically, the spring-test module declares all core default TestExecutionListeners under the org.springframework.test.context.TestExecutionListener key in its META-INF/spring.factories properties file. Third-party frameworks and developers can contribute their own TestExecutionListeners to the list of default listeners in the same manner via their own META-INF/spring.factories properties file.

Ordering TestExecutionListeners

When the TestContext framework discovers default TestExecutionListeners via the aforementioned SpringFactoriesLoader mechanism, the instantiated listeners are sorted using Spring’s AnnotationAwareOrderComparator which honors Spring’s Ordered interface and @Order annotation for ordering. AbstractTestExecutionListener and all default TestExecutionListeners provided by Spring implement Ordered with appropriate values. Third-party frameworks and developers should therefore make sure that their default TestExecutionListeners are registered in the proper order by implementing Ordered or declaring @Order. Consult the javadocs for the getOrder() methods of the core default TestExecutionListeners for details on what values are assigned to each core listener.

Merging TestExecutionListeners

If a custom TestExecutionListener is registered via @TestExecutionListeners, the default listeners will not be registered. In most common testing scenarios, this effectively forces the developer to manually declare all default listeners in addition to any custom listeners. The following listing demonstrates this style of configuration.

@ContextConfiguration

@TestExecutionListeners({

MyCustomTestExecutionListener.class,

ServletTestExecutionListener.class,

DirtiesContextBeforeModesTestExecutionListener.class,

DependencyInjectionTestExecutionListener.class,

DirtiesContextTestExecutionListener.class,

TransactionalTestExecutionListener.class,

SqlScriptsTestExecutionListener.class

})

public class MyTest {

// class body...

}

The challenge with this approach is that it requires that the developer know exactly which listeners are registered by default. Moreover, the set of default listeners can change from release to release — for example, SqlScriptsTestExecutionListener was introduced in Spring Framework 4.1, and DirtiesContextBeforeModesTestExecutionListener was introduced in Spring Framework 4.2. Furthermore, third-party frameworks like Spring Security register their own default TestExecutionListeners via the aforementioned automatic discovery mechanism.

To avoid having to be aware of and re-declare all default listeners, the mergeMode attribute of @TestExecutionListeners can be set to MergeMode.MERGE\_WITH\_DEFAULTS. MERGE\_WITH\_DEFAULTS indicates that locally declared listeners should be merged with the default listeners. The merging algorithm ensures that duplicates are removed from the list and that the resulting set of merged listeners is sorted according to the semantics of AnnotationAwareOrderComparator as described in Ordering TestExecutionListeners. If a listener implements Ordered or is annotated with @Order it can influence the position in which it is merged with the defaults; otherwise, locally declared listeners will simply be appended to the list of default listeners when merged.

For example, if the MyCustomTestExecutionListener class in the previous example configures its order value (for example, 500) to be less than the order of the ServletTestExecutionListener (which happens to be 1000), the MyCustomTestExecutionListener can then be automatically merged with the list of defaults in front of the ServletTestExecutionListener, and the previous example could be replaced with the following.

@ContextConfiguration

@TestExecutionListeners(

listeners = MyCustomTestExecutionListener.class,

mergeMode = MERGE\_WITH\_DEFAULTS

)

public class MyTest {

// class body...

}

3.5.4. Context management

Each TestContext provides context management and caching support for the test instance it is responsible for. Test instances do not automatically receive access to the configured ApplicationContext. However, if a test class implements the ApplicationContextAware interface, a reference to the ApplicationContext is supplied to the test instance. Note that AbstractJUnit4SpringContextTests and AbstractTestNGSpringContextTests implement ApplicationContextAware and therefore provide access to the ApplicationContext automatically.

@Autowired ApplicationContext

As an alternative to implementing the ApplicationContextAware interface, you can inject the application context for your test class through the @Autowired annotation on either a field or setter method. For example:

@RunWith(SpringRunner.class)

@ContextConfiguration

public class MyTest {

@Autowired

private ApplicationContext applicationContext;

// class body...

}

Similarly, if your test is configured to load a WebApplicationContext, you can inject the web application context into your test as follows:

@RunWith(SpringRunner.class)

@WebAppConfiguration

@ContextConfiguration

public class MyWebAppTest {

@Autowired

private WebApplicationContext wac;

// class body...

}

Dependency injection via @Autowired is provided by the DependencyInjectionTestExecutionListener which is configured by default (see Dependency injection of test fixtures).

Test classes that use the TestContext framework do not need to extend any particular class or implement a specific interface to configure their application context. Instead, configuration is achieved simply by declaring the @ContextConfiguration annotation at the class level. If your test class does not explicitly declare application context resource locations or annotated classes, the configured ContextLoader determines how to load a context from a default location or default configuration classes. In addition to context resource locations and annotated classes, an application context can also be configured via application context initializers.

The following sections explain how to configure an ApplicationContext via XML configuration files, Groovy scripts, annotated classes (typically @Configuration classes), or context initializers using Spring’s @ContextConfiguration annotation. Alternatively, you can implement and configure your own custom SmartContextLoader for advanced use cases.

Context configuration with XML resources

To load an ApplicationContext for your tests using XML configuration files, annotate your test class with @ContextConfiguration and configure the locations attribute with an array that contains the resource locations of XML configuration metadata. A plain or relative path — for example "context.xml" — will be treated as a classpath resource that is relative to the package in which the test class is defined. A path starting with a slash is treated as an absolute classpath location, for example "/org/example/config.xml". A path which represents a resource URL (i.e., a path prefixed with classpath:, file:, http:, etc.) will be used as is.

@RunWith(SpringRunner.class)

// ApplicationContext will be loaded from "/app-config.xml" and

// "/test-config.xml" in the root of the classpath

@ContextConfiguration(locations={"/app-config.xml", "/test-config.xml"})

public class MyTest {

// class body...

}

@ContextConfiguration supports an alias for the locations attribute through the standard Java value attribute. Thus, if you do not need to declare additional attributes in @ContextConfiguration, you can omit the declaration of the locations attribute name and declare the resource locations by using the shorthand format demonstrated in the following example.

@RunWith(SpringRunner.class)

@ContextConfiguration({"/app-config.xml", "/test-config.xml"})

public class MyTest {

// class body...

}

If you omit both the locations and value attributes from the @ContextConfiguration annotation, the TestContext framework will attempt to detect a default XML resource location. Specifically, GenericXmlContextLoader and GenericXmlWebContextLoader detect a default location based on the name of the test class. If your class is named com.example.MyTest, GenericXmlContextLoader loads your application context from "classpath:com/example/MyTest-context.xml".

package com.example;

@RunWith(SpringRunner.class)

// ApplicationContext will be loaded from

// "classpath:com/example/MyTest-context.xml"

@ContextConfiguration

public class MyTest {

// class body...

}

Context configuration with Groovy scripts

To load an ApplicationContext for your tests using Groovy scripts that utilize the Groovy Bean Definition DSL, annotate your test class with @ContextConfiguration and configure the locations or value attribute with an array that contains the resource locations of Groovy scripts. Resource lookup semantics for Groovy scripts are the same as those described for XML configuration files.

Enabling Groovy script support

Support for using Groovy scripts to load an ApplicationContext in the Spring TestContext Framework is enabled automatically if Groovy is on the classpath.

@RunWith(SpringRunner.class)

// ApplicationContext will be loaded from "/AppConfig.groovy" and

// "/TestConfig.groovy" in the root of the classpath

@ContextConfiguration({"/AppConfig.groovy", "/TestConfig.Groovy"})

public class MyTest {

// class body...

}

If you omit both the locations and value attributes from the @ContextConfiguration annotation, the TestContext framework will attempt to detect a default Groovy script. Specifically, GenericGroovyXmlContextLoader and GenericGroovyXmlWebContextLoader detect a default location based on the name of the test class. If your class is named com.example.MyTest, the Groovy context loader will load your application context from "classpath:com/example/MyTestContext.groovy".

package com.example;

@RunWith(SpringRunner.class)

// ApplicationContext will be loaded from

// "classpath:com/example/MyTestContext.groovy"

@ContextConfiguration

public class MyTest {

// class body...

}

Declaring XML config and Groovy scripts simultaneously

Both XML configuration files and Groovy scripts can be declared simultaneously via the locations or value attribute of @ContextConfiguration. If the path to a configured resource location ends with .xml it will be loaded using an XmlBeanDefinitionReader; otherwise it will be loaded using a GroovyBeanDefinitionReader.

The following listing demonstrates how to combine both in an integration test.

@RunWith(SpringRunner.class)

// ApplicationContext will be loaded from

// "/app-config.xml" and "/TestConfig.groovy"

@ContextConfiguration({ "/app-config.xml", "/TestConfig.groovy" })

public class MyTest {

// class body...

}

Context configuration with annotated classes

To load an ApplicationContext for your tests using annotated classes (see Java-based container configuration), annotate your test class with @ContextConfiguration and configure the classes attribute with an array that contains references to annotated classes.

@RunWith(SpringRunner.class)

// ApplicationContext will be loaded from AppConfig and TestConfig

@ContextConfiguration(classes = {AppConfig.class, TestConfig.class})

public class MyTest {

// class body...

}

Annotated Classes

The term annotated class can refer to any of the following.

A class annotated with @Configuration

A component (i.e., a class annotated with @Component, @Service, @Repository, etc.)

A JSR-330 compliant class that is annotated with javax.inject annotations

Any other class that contains @Bean-methods

Consult the javadocs of @Configuration and @Bean for further information regarding the configuration and semantics of annotated classes, paying special attention to the discussion of `@Bean` Lite Mode.

If you omit the classes attribute from the @ContextConfiguration annotation, the TestContext framework will attempt to detect the presence of default configuration classes. Specifically, AnnotationConfigContextLoader and AnnotationConfigWebContextLoader will detect all static nested classes of the test class that meet the requirements for configuration class implementations as specified in the @Configuration javadocs. In the following example, the OrderServiceTest class declares a static nested configuration class named Config that will be automatically used to load the ApplicationContext for the test class. Note that the name of the configuration class is arbitrary. In addition, a test class can contain more than one static nested configuration class if desired.

@RunWith(SpringRunner.class)

// ApplicationContext will be loaded from the

// static nested Config class

@ContextConfiguration

public class OrderServiceTest {

@Configuration

static class Config {

// this bean will be injected into the OrderServiceTest class

@Bean

public OrderService orderService() {

OrderService orderService = new OrderServiceImpl();

// set properties, etc.

return orderService;

}

}

@Autowired

private OrderService orderService;

@Test

public void testOrderService() {

// test the orderService

}

}

Mixing XML, Groovy scripts, and annotated classes

It may sometimes be desirable to mix XML configuration files, Groovy scripts, and annotated classes (i.e., typically @Configuration classes) to configure an ApplicationContext for your tests. For example, if you use XML configuration in production, you may decide that you want to use @Configuration classes to configure specific Spring-managed components for your tests, or vice versa.

Furthermore, some third-party frameworks (like Spring Boot) provide first-class support for loading an ApplicationContext from different types of resources simultaneously (e.g., XML configuration files, Groovy scripts, and @Configuration classes). The Spring Framework historically has not supported this for standard deployments. Consequently, most of the SmartContextLoader implementations that the Spring Framework delivers in the spring-test module support only one resource type per test context; however, this does not mean that you cannot use both. One exception to the general rule is that the GenericGroovyXmlContextLoader and GenericGroovyXmlWebContextLoader support both XML configuration files and Groovy scripts simultaneously. Furthermore, third-party frameworks may choose to support the declaration of both locations and classes via @ContextConfiguration, and with the standard testing support in the TestContext framework, you have the following options.

If you want to use resource locations (e.g., XML or Groovy) and @Configuration classes to configure your tests, you will have to pick one as the entry point, and that one will have to include or import the other. For example, in XML or Groovy scripts you can include @Configuration classes via component scanning or define them as normal Spring beans; whereas, in a @Configuration class you can use @ImportResource to import XML configuration files or Groovy scripts. Note that this behavior is semantically equivalent to how you configure your application in production: in production configuration you will define either a set of XML or Groovy resource locations or a set of @Configuration classes that your production ApplicationContext will be loaded from, but you still have the freedom to include or import the other type of configuration.

Context configuration with context initializers

To configure an ApplicationContext for your tests using context initializers, annotate your test class with @ContextConfiguration and configure the initializers attribute with an array that contains references to classes that implement ApplicationContextInitializer. The declared context initializers will then be used to initialize the ConfigurableApplicationContext that is loaded for your tests. Note that the concrete ConfigurableApplicationContext type supported by each declared initializer must be compatible with the type of ApplicationContext created by the SmartContextLoader in use (i.e., typically a GenericApplicationContext). Furthermore, the order in which the initializers are invoked depends on whether they implement Spring’s Ordered interface or are annotated with Spring’s @Order annotation or the standard @Priority annotation.

@RunWith(SpringRunner.class)

// ApplicationContext will be loaded from TestConfig

// and initialized by TestAppCtxInitializer

@ContextConfiguration(

classes = TestConfig.class,

initializers = TestAppCtxInitializer.class)

public class MyTest {

// class body...

}

It is also possible to omit the declaration of XML configuration files, Groovy scripts, or annotated classes in @ContextConfiguration entirely and instead declare only ApplicationContextInitializer classes which are then responsible for registering beans in the context — for example, by programmatically loading bean definitions from XML files or configuration classes.

@RunWith(SpringRunner.class)

// ApplicationContext will be initialized by EntireAppInitializer

// which presumably registers beans in the context

@ContextConfiguration(initializers = EntireAppInitializer.class)

public class MyTest {

// class body...

}

Context configuration inheritance

@ContextConfiguration supports boolean inheritLocations and inheritInitializers attributes that denote whether resource locations or annotated classes and context initializers declared by superclasses should be inherited. The default value for both flags is true. This means that a test class inherits the resource locations or annotated classes as well as the context initializers declared by any superclasses. Specifically, the resource locations or annotated classes for a test class are appended to the list of resource locations or annotated classes declared by superclasses. Similarly, the initializers for a given test class will be added to the set of initializers defined by test superclasses. Thus, subclasses have the option of extending the resource locations, annotated classes, or context initializers.

If the inheritLocations or inheritInitializers attribute in @ContextConfiguration is set to false, the resource locations or annotated classes and the context initializers, respectively, for the test class shadow and effectively replace the configuration defined by superclasses.

In the following example that uses XML resource locations, the ApplicationContext for ExtendedTest will be loaded from "base-config.xml" and "extended-config.xml", in that order. Beans defined in "extended-config.xml" may therefore override (i.e., replace) those defined in "base-config.xml".

@RunWith(SpringRunner.class)

// ApplicationContext will be loaded from "/base-config.xml"

// in the root of the classpath

@ContextConfiguration("/base-config.xml")

public class BaseTest {

// class body...

}

// ApplicationContext will be loaded from "/base-config.xml" and

// "/extended-config.xml" in the root of the classpath

@ContextConfiguration("/extended-config.xml")

public class ExtendedTest extends BaseTest {

// class body...

}

Similarly, in the following example that uses annotated classes, the ApplicationContext for ExtendedTest will be loaded from the BaseConfig and ExtendedConfig classes, in that order. Beans defined in ExtendedConfig may therefore override (i.e., replace) those defined in BaseConfig.

@RunWith(SpringRunner.class)

// ApplicationContext will be loaded from BaseConfig

@ContextConfiguration(classes = BaseConfig.class)

public class BaseTest {

// class body...

}

// ApplicationContext will be loaded from BaseConfig and ExtendedConfig

@ContextConfiguration(classes = ExtendedConfig.class)

public class ExtendedTest extends BaseTest {

// class body...

}

In the following example that uses context initializers, the ApplicationContext for ExtendedTest will be initialized using BaseInitializer and ExtendedInitializer. Note, however, that the order in which the initializers are invoked depends on whether they implement Spring’s Ordered interface or are annotated with Spring’s @Order annotation or the standard @Priority annotation.

@RunWith(SpringRunner.class)

// ApplicationContext will be initialized by BaseInitializer

@ContextConfiguration(initializers = BaseInitializer.class)

public class BaseTest {

// class body...

}

// ApplicationContext will be initialized by BaseInitializer

// and ExtendedInitializer

@ContextConfiguration(initializers = ExtendedInitializer.class)

public class ExtendedTest extends BaseTest {

// class body...

}

Context configuration with environment profiles

Spring 3.1 introduced first-class support in the framework for the notion of environments and profiles (a.k.a., bean definition profiles), and integration tests can be configured to activate particular bean definition profiles for various testing scenarios. This is achieved by annotating a test class with the @ActiveProfiles annotation and supplying a list of profiles that should be activated when loading the ApplicationContext for the test.

@ActiveProfiles may be used with any implementation of the new SmartContextLoader SPI, but @ActiveProfiles is not supported with implementations of the older ContextLoader SPI.

Let’s take a look at some examples with XML configuration and @Configuration classes.

<!-- app-config.xml -->

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:jdbc="http://www.springframework.org/schema/jdbc"

xmlns:jee="http://www.springframework.org/schema/jee"

xsi:schemaLocation="...">

<bean id="transferService"

class="com.bank.service.internal.DefaultTransferService">

<constructor-arg ref="accountRepository"/>

<constructor-arg ref="feePolicy"/>

</bean>

<bean id="accountRepository"

class="com.bank.repository.internal.JdbcAccountRepository">

<constructor-arg ref="dataSource"/>

</bean>

<bean id="feePolicy"

class="com.bank.service.internal.ZeroFeePolicy"/>

<beans profile="dev">

<jdbc:embedded-database id="dataSource">

<jdbc:script

location="classpath:com/bank/config/sql/schema.sql"/>

<jdbc:script

location="classpath:com/bank/config/sql/test-data.sql"/>

</jdbc:embedded-database>

</beans>

<beans profile="production">

<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>

</beans>

<beans profile="default">

<jdbc:embedded-database id="dataSource">

<jdbc:script

location="classpath:com/bank/config/sql/schema.sql"/>

</jdbc:embedded-database>

</beans>

</beans>

package com.bank.service;

@RunWith(SpringRunner.class)

// ApplicationContext will be loaded from "classpath:/app-config.xml"

@ContextConfiguration("/app-config.xml")

@ActiveProfiles("dev")

public class TransferServiceTest {

@Autowired

private TransferService transferService;

@Test

public void testTransferService() {

// test the transferService

}

}

When TransferServiceTest is run, its ApplicationContext will be loaded from the app-config.xml configuration file in the root of the classpath. If you inspect app-config.xml you’ll notice that the accountRepository bean has a dependency on a dataSource bean; however, dataSource is not defined as a top-level bean. Instead, dataSource is defined three times: in the production profile, the dev profile, and the default profile.

By annotating TransferServiceTest with @ActiveProfiles("dev") we instruct the Spring TestContext Framework to load the ApplicationContext with the active profiles set to {"dev"}. As a result, an embedded database will be created and populated with test data, and the accountRepository bean will be wired with a reference to the development DataSource. And that’s likely what we want in an integration test.

It is sometimes useful to assign beans to a default profile. Beans within the default profile are only included when no other profile is specifically activated. This can be used to define fallback beans to be used in the application’s default state. For example, you may explicitly provide a data source for dev and production profiles, but define an in-memory data source as a default when neither of these is active.

The following code listings demonstrate how to implement the same configuration and integration test but using @Configuration classes instead of XML.

@Configuration

@Profile("dev")

public class StandaloneDataConfig {

@Bean

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.addScript("classpath:com/bank/config/sql/test-data.sql")

.build();

}

}

@Configuration

@Profile("production")

public class JndiDataConfig {

@Bean(destroyMethod="")

public DataSource dataSource() throws Exception {

Context ctx = new InitialContext();

return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");

}

}

@Configuration

@Profile("default")

public class DefaultDataConfig {

@Bean

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.HSQL)

.addScript("classpath:com/bank/config/sql/schema.sql")

.build();

}

}

@Configuration

public class TransferServiceConfig {

@Autowired DataSource dataSource;

@Bean

public TransferService transferService() {

return new DefaultTransferService(accountRepository(), feePolicy());

}

@Bean

public AccountRepository accountRepository() {

return new JdbcAccountRepository(dataSource);

}

@Bean

public FeePolicy feePolicy() {

return new ZeroFeePolicy();

}

}

package com.bank.service;

@RunWith(SpringRunner.class)

@ContextConfiguration(classes = {

TransferServiceConfig.class,

StandaloneDataConfig.class,

JndiDataConfig.class,

DefaultDataConfig.class})

@ActiveProfiles("dev")

public class TransferServiceTest {

@Autowired

private TransferService transferService;

@Test

public void testTransferService() {

// test the transferService

}

}

In this variation, we have split the XML configuration into four independent @Configuration classes:

TransferServiceConfig: acquires a dataSource via dependency injection using @Autowired

StandaloneDataConfig: defines a dataSource for an embedded database suitable for developer tests

JndiDataConfig: defines a dataSource that is retrieved from JNDI in a production environment

DefaultDataConfig: defines a dataSource for a default embedded database in case no profile is active

As with the XML-based configuration example, we still annotate TransferServiceTest with @ActiveProfiles("dev"), but this time we specify all four configuration classes via the @ContextConfiguration annotation. The body of the test class itself remains completely unchanged.

It is often the case that a single set of profiles is used across multiple test classes within a given project. Thus, to avoid duplicate declarations of the @ActiveProfiles annotation it is possible to declare @ActiveProfiles once on a base class, and subclasses will automatically inherit the @ActiveProfiles configuration from the base class. In the following example, the declaration of @ActiveProfiles (as well as other annotations) has been moved to an abstract superclass, AbstractIntegrationTest.

package com.bank.service;

@RunWith(SpringRunner.class)

@ContextConfiguration(classes = {

TransferServiceConfig.class,

StandaloneDataConfig.class,

JndiDataConfig.class,

DefaultDataConfig.class})

@ActiveProfiles("dev")

public abstract class AbstractIntegrationTest {

}

package com.bank.service;

// "dev" profile inherited from superclass

public class TransferServiceTest extends AbstractIntegrationTest {

@Autowired

private TransferService transferService;

@Test

public void testTransferService() {

// test the transferService

}

}

@ActiveProfiles also supports an inheritProfiles attribute that can be used to disable the inheritance of active profiles.

package com.bank.service;

// "dev" profile overridden with "production"

@ActiveProfiles(profiles = "production", inheritProfiles = false)

public class ProductionTransferServiceTest extends AbstractIntegrationTest {

// test body

}

Furthermore, it is sometimes necessary to resolve active profiles for tests programmatically instead of declaratively — for example, based on:

the current operating system

whether tests are being executed on a continuous integration build server

the presence of certain environment variables

the presence of custom class-level annotations

etc.

To resolve active bean definition profiles programmatically, simply implement a custom ActiveProfilesResolver and register it via the resolver attribute of @ActiveProfiles. The following example demonstrates how to implement and register a custom OperatingSystemActiveProfilesResolver. For further information, refer to the corresponding javadocs.

package com.bank.service;

// "dev" profile overridden programmatically via a custom resolver

@ActiveProfiles(

resolver = OperatingSystemActiveProfilesResolver.class,

inheritProfiles = false)

public class TransferServiceTest extends AbstractIntegrationTest {

// test body

}

package com.bank.service.test;

public class OperatingSystemActiveProfilesResolver implements ActiveProfilesResolver {

@Override

String[] resolve(Class<?> testClass) {

String profile = ...;

// determine the value of profile based on the operating system

return new String[] {profile};

}

}

Context configuration with test property sources

Spring 3.1 introduced first-class support in the framework for the notion of an environment with a hierarchy of property sources, and since Spring 4.1 integration tests can be configured with test-specific property sources. In contrast to the @PropertySource annotation used on @Configuration classes, the @TestPropertySource annotation can be declared on a test class to declare resource locations for test properties files or inlined properties. These test property sources will be added to the set of PropertySources in the Environment for the ApplicationContext loaded for the annotated integration test.

@TestPropertySource may be used with any implementation of the SmartContextLoader SPI, but @TestPropertySource is not supported with implementations of the older ContextLoader SPI.

Implementations of SmartContextLoader gain access to merged test property source values via the getPropertySourceLocations() and getPropertySourceProperties() methods in MergedContextConfiguration.

Declaring test property sources

Test properties files can be configured via the locations or value attribute of @TestPropertySource as shown in the following example.

Both traditional and XML-based properties file formats are supported — for example, "classpath:/com/example/test.properties" or "file:///path/to/file.xml".

Each path will be interpreted as a Spring Resource. A plain path — for example, "test.properties" — will be treated as a classpath resource that is relative to the package in which the test class is defined. A path starting with a slash will be treated as an absolute classpath resource, for example: "/org/example/test.xml". A path which references a URL (e.g., a path prefixed with classpath:, file:, http:, etc.) will be loaded using the specified resource protocol. Resource location wildcards (e.g. \*/.properties) are not permitted: each location must evaluate to exactly one .properties or .xml resource.

@ContextConfiguration

@TestPropertySource("/test.properties")

public class MyIntegrationTests {

// class body...

}

Inlined properties in the form of key-value pairs can be configured via the properties attribute of @TestPropertySource as shown in the following example. All key-value pairs will be added to the enclosing Environment as a single test PropertySource with the highest precedence.

The supported syntax for key-value pairs is the same as the syntax defined for entries in a Java properties file:

"key=value"

"key:value"

"key value"

@ContextConfiguration

@TestPropertySource(properties = {"timezone = GMT", "port: 4242"})

public class MyIntegrationTests {

// class body...

}

Default properties file detection

If @TestPropertySource is declared as an empty annotation (i.e., without explicit values for the locations or properties attributes), an attempt will be made to detect a default properties file relative to the class that declared the annotation. For example, if the annotated test class is com.example.MyTest, the corresponding default properties file is "classpath:com/example/MyTest.properties". If the default cannot be detected, an IllegalStateException will be thrown.

Precedence

Test property sources have higher precedence than those loaded from the operating system’s environment or Java system properties as well as property sources added by the application declaratively via @PropertySource or programmatically. Thus, test property sources can be used to selectively override properties defined in system and application property sources. Furthermore, inlined properties have higher precedence than properties loaded from resource locations.

In the following example, the timezone and port properties as well as any properties defined in "/test.properties" will override any properties of the same name that are defined in system and application property sources. Furthermore, if the "/test.properties" file defines entries for the timezone and port properties those will be overridden by the inlined properties declared via the properties attribute.

@ContextConfiguration

@TestPropertySource(

locations = "/test.properties",

properties = {"timezone = GMT", "port: 4242"}

)

public class MyIntegrationTests {

// class body...

}

Inheriting and overriding test property sources

@TestPropertySource supports boolean inheritLocations and inheritProperties attributes that denote whether resource locations for properties files and inlined properties declared by superclasses should be inherited. The default value for both flags is true. This means that a test class inherits the locations and inlined properties declared by any superclasses. Specifically, the locations and inlined properties for a test class are appended to the locations and inlined properties declared by superclasses. Thus, subclasses have the option of extending the locations and inlined properties. Note that properties that appear later will shadow (i.e.., override) properties of the same name that appear earlier. In addition, the aforementioned precedence rules apply for inherited test property sources as well.

If the inheritLocations or inheritProperties attribute in @TestPropertySource is set to false, the locations or inlined properties, respectively, for the test class shadow and effectively replace the configuration defined by superclasses.

In the following example, the ApplicationContext for BaseTest will be loaded using only the "base.properties" file as a test property source. In contrast, the ApplicationContext for ExtendedTest will be loaded using the "base.properties" and "extended.properties" files as test property source locations.

@TestPropertySource("base.properties")

@ContextConfiguration

public class BaseTest {

// ...

}

@TestPropertySource("extended.properties")

@ContextConfiguration

public class ExtendedTest extends BaseTest {

// ...

}

In the following example, the ApplicationContext for BaseTest will be loaded using only the inlined key1 property. In contrast, the ApplicationContext for ExtendedTest will be loaded using the inlined key1 and key2 properties.

@TestPropertySource(properties = "key1 = value1")

@ContextConfiguration

public class BaseTest {

// ...

}

@TestPropertySource(properties = "key2 = value2")

@ContextConfiguration

public class ExtendedTest extends BaseTest {

// ...

}

Loading a WebApplicationContext

Spring 3.2 introduced support for loading a WebApplicationContext in integration tests. To instruct the TestContext framework to load a WebApplicationContext instead of a standard ApplicationContext, simply annotate the respective test class with @WebAppConfiguration.

The presence of @WebAppConfiguration on your test class instructs the TestContext framework (TCF) that a WebApplicationContext (WAC) should be loaded for your integration tests. In the background the TCF makes sure that a MockServletContext is created and supplied to your test’s WAC. By default the base resource path for your MockServletContext will be set to "src/main/webapp". This is interpreted as a path relative to the root of your JVM (i.e., normally the path to your project). If you’re familiar with the directory structure of a web application in a Maven project, you’ll know that "src/main/webapp" is the default location for the root of your WAR. If you need to override this default, simply provide an alternate path to the @WebAppConfiguration annotation (e.g., @WebAppConfiguration("src/test/webapp")). If you wish to reference a base resource path from the classpath instead of the file system, just use Spring’s classpath: prefix.

Please note that Spring’s testing support for WebApplicationContexts is on par with its support for standard ApplicationContexts. When testing with a WebApplicationContext you are free to declare XML configuration files, Groovy scripts, or @Configuration classes via @ContextConfiguration. You are of course also free to use any other test annotations such as @ActiveProfiles, @TestExecutionListeners, @Sql, @Rollback, etc.

The following examples demonstrate some of the various configuration options for loading a WebApplicationContext.

Conventions

@RunWith(SpringRunner.class)

// defaults to "file:src/main/webapp"

@WebAppConfiguration

// detects "WacTests-context.xml" in same package

// or static nested @Configuration class

@ContextConfiguration

public class WacTests {

//...

}

The above example demonstrates the TestContext framework’s support for convention over configuration. If you annotate a test class with @WebAppConfiguration without specifying a resource base path, the resource path will effectively default to "file:src/main/webapp". Similarly, if you declare @ContextConfiguration without specifying resource locations, annotated classes, or context initializers, Spring will attempt to detect the presence of your configuration using conventions (i.e., "WacTests-context.xml" in the same package as the WacTests class or static nested @Configuration classes).

Default resource semantics

@RunWith(SpringRunner.class)

// file system resource

@WebAppConfiguration("webapp")

// classpath resource

@ContextConfiguration("/spring/test-servlet-config.xml")

public class WacTests {

//...

}

This example demonstrates how to explicitly declare a resource base path with @WebAppConfiguration and an XML resource location with @ContextConfiguration. The important thing to note here is the different semantics for paths with these two annotations. By default, @WebAppConfiguration resource paths are file system based; whereas, @ContextConfiguration resource locations are classpath based.

Explicit resource semantics

@RunWith(SpringRunner.class)

// classpath resource

@WebAppConfiguration("classpath:test-web-resources")

// file system resource

@ContextConfiguration("file:src/main/webapp/WEB-INF/servlet-config.xml")

public class WacTests {

//...

}

In this third example, we see that we can override the default resource semantics for both annotations by specifying a Spring resource prefix. Contrast the comments in this example with the previous example.

Working with Web Mocks

To provide comprehensive web testing support, Spring 3.2 introduced a ServletTestExecutionListener that is enabled by default. When testing against a WebApplicationContext this TestExecutionListener sets up default thread-local state via Spring Web’s RequestContextHolder before each test method and creates a MockHttpServletRequest, MockHttpServletResponse, and ServletWebRequest based on the base resource path configured via @WebAppConfiguration. ServletTestExecutionListener also ensures that the MockHttpServletResponse and ServletWebRequest can be injected into the test instance, and once the test is complete it cleans up thread-local state.

Once you have a WebApplicationContext loaded for your test you might find that you need to interact with the web mocks — for example, to set up your test fixture or to perform assertions after invoking your web component. The following example demonstrates which mocks can be autowired into your test instance. Note that the WebApplicationContext and MockServletContext are both cached across the test suite; whereas, the other mocks are managed per test method by the ServletTestExecutionListener.

Injecting mocks

@WebAppConfiguration

@ContextConfiguration

public class WacTests {

@Autowired

WebApplicationContext wac; // cached

@Autowired

MockServletContext servletContext; // cached

@Autowired

MockHttpSession session;

@Autowired

MockHttpServletRequest request;

@Autowired

MockHttpServletResponse response;

@Autowired

ServletWebRequest webRequest;

//...

}

Context caching

Once the TestContext framework loads an ApplicationContext (or WebApplicationContext) for a test, that context will be cached and reused for all subsequent tests that declare the same unique context configuration within the same test suite. To understand how caching works, it is important to understand what is meant by unique and test suite.

An ApplicationContext can be uniquely identified by the combination of configuration parameters that is used to load it. Consequently, the unique combination of configuration parameters is used to generate a key under which the context is cached. The TestContext framework uses the following configuration parameters to build the context cache key:

locations (from @ContextConfiguration)

classes (from @ContextConfiguration)

contextInitializerClasses (from @ContextConfiguration)

contextCustomizers (from ContextCustomizerFactory)

contextLoader (from @ContextConfiguration)

parent (from @ContextHierarchy)

activeProfiles (from @ActiveProfiles)

propertySourceLocations (from @TestPropertySource)

propertySourceProperties (from @TestPropertySource)

resourceBasePath (from @WebAppConfiguration)

For example, if TestClassA specifies {"app-config.xml", "test-config.xml"} for the locations (or value) attribute of @ContextConfiguration, the TestContext framework will load the corresponding ApplicationContext and store it in a static context cache under a key that is based solely on those locations. So if TestClassB also defines {"app-config.xml", "test-config.xml"} for its locations (either explicitly or implicitly through inheritance) but does not define @WebAppConfiguration, a different ContextLoader, different active profiles, different context initializers, different test property sources, or a different parent context, then the same ApplicationContext will be shared by both test classes. This means that the setup cost for loading an application context is incurred only once (per test suite), and subsequent test execution is much faster.

Test suites and forked processes

The Spring TestContext framework stores application contexts in a static cache. This means that the context is literally stored in a static variable. In other words, if tests execute in separate processes the static cache will be cleared between each test execution, and this will effectively disable the caching mechanism.

To benefit from the caching mechanism, all tests must run within the same process or test suite. This can be achieved by executing all tests as a group within an IDE. Similarly, when executing tests with a build framework such as Ant, Maven, or Gradle it is important to make sure that the build framework does not fork between tests. For example, if the forkMode for the Maven Surefire plug-in is set to always or pertest, the TestContext framework will not be able to cache application contexts between test classes and the build process will run significantly slower as a result.

Since Spring Framework 4.3, the size of the context cache is bounded with a default maximum size of 32. Whenever the maximum size is reached, a least recently used (LRU) eviction policy is used to evict and close stale contexts. The maximum size can be configured from the command line or a build script by setting a JVM system property named spring.test.context.cache.maxSize. As an alternative, the same property can be set programmatically via the SpringProperties API.

Since having a large number of application contexts loaded within a given test suite can cause the suite to take an unnecessarily long time to execute, it is often beneficial to know exactly how many contexts have been loaded and cached. To view the statistics for the underlying context cache, simply set the log level for the org.springframework.test.context.cache logging category to DEBUG.

In the unlikely case that a test corrupts the application context and requires reloading — for example, by modifying a bean definition or the state of an application object — you can annotate your test class or test method with @DirtiesContext (see the discussion of @DirtiesContext in Spring Testing Annotations). This instructs Spring to remove the context from the cache and rebuild the application context before executing the next test. Note that support for the @DirtiesContext annotation is provided by the DirtiesContextBeforeModesTestExecutionListener and the DirtiesContextTestExecutionListener which are enabled by default.

Context hierarchies

When writing integration tests that rely on a loaded Spring ApplicationContext, it is often sufficient to test against a single context; however, there are times when it is beneficial or even necessary to test against a hierarchy of ApplicationContexts. For example, if you are developing a Spring MVC web application you will typically have a root WebApplicationContext loaded via Spring’s ContextLoaderListener and a child WebApplicationContext loaded via Spring’s DispatcherServlet. This results in a parent-child context hierarchy where shared components and infrastructure configuration are declared in the root context and consumed in the child context by web-specific components. Another use case can be found in Spring Batch applications where you often have a parent context that provides configuration for shared batch infrastructure and a child context for the configuration of a specific batch job.

Since Spring Framework 3.2.2, it is possible to write integration tests that use context hierarchies by declaring context configuration via the @ContextHierarchy annotation, either on an individual test class or within a test class hierarchy. If a context hierarchy is declared on multiple classes within a test class hierarchy it is also possible to merge or override the context configuration for a specific, named level in the context hierarchy. When merging configuration for a given level in the hierarchy the configuration resource type (i.e., XML configuration files or annotated classes) must be consistent; otherwise, it is perfectly acceptable to have different levels in a context hierarchy configured using different resource types.

The following JUnit 4 based examples demonstrate common configuration scenarios for integration tests that require the use of context hierarchies.

Single test class with context hierarchy

ControllerIntegrationTests represents a typical integration testing scenario for a Spring MVC web application by declaring a context hierarchy consisting of two levels, one for the root WebApplicationContext (loaded using the TestAppConfig @Configuration class) and one for the dispatcher servlet WebApplicationContext (loaded using the WebConfig @Configuration class). The WebApplicationContext that is autowired into the test instance is the one for the child context (i.e., the lowest context in the hierarchy).

@RunWith(SpringRunner.class)

@WebAppConfiguration

@ContextHierarchy({

@ContextConfiguration(classes = TestAppConfig.class),

@ContextConfiguration(classes = WebConfig.class)

})

public class ControllerIntegrationTests {

@Autowired

private WebApplicationContext wac;

// ...

}

Class hierarchy with implicit parent context

The following test classes define a context hierarchy within a test class hierarchy. AbstractWebTests declares the configuration for a root WebApplicationContext in a Spring-powered web application. Note, however, that AbstractWebTests does not declare @ContextHierarchy; consequently, subclasses of AbstractWebTests can optionally participate in a context hierarchy or simply follow the standard semantics for @ContextConfiguration. SoapWebServiceTests and RestWebServiceTests both extend AbstractWebTests and define a context hierarchy via @ContextHierarchy. The result is that three application contexts will be loaded (one for each declaration of @ContextConfiguration), and the application context loaded based on the configuration in AbstractWebTests will be set as the parent context for each of the contexts loaded for the concrete subclasses.

@RunWith(SpringRunner.class)

@WebAppConfiguration

@ContextConfiguration("file:src/main/webapp/WEB-INF/applicationContext.xml")

public abstract class AbstractWebTests {}

@ContextHierarchy(@ContextConfiguration("/spring/soap-ws-config.xml")

public class SoapWebServiceTests extends AbstractWebTests {}

@ContextHierarchy(@ContextConfiguration("/spring/rest-ws-config.xml")

public class RestWebServiceTests extends AbstractWebTests {}

Class hierarchy with merged context hierarchy configuration

The following classes demonstrate the use of named hierarchy levels in order to merge the configuration for specific levels in a context hierarchy. BaseTests defines two levels in the hierarchy, parent and child. ExtendedTests extends BaseTests and instructs the Spring TestContext Framework to merge the context configuration for the child hierarchy level, simply by ensuring that the names declared via the name attribute in @ContextConfiguration are both "child". The result is that three application contexts will be loaded: one for "/app-config.xml", one for "/user-config.xml", and one for {"/user-config.xml", "/order-config.xml"}. As with the previous example, the application context loaded from "/app-config.xml" will be set as the parent context for the contexts loaded from "/user-config.xml" and {"/user-config.xml", "/order-config.xml"}.

@RunWith(SpringRunner.class)

@ContextHierarchy({

@ContextConfiguration(name = "parent", locations = "/app-config.xml"),

@ContextConfiguration(name = "child", locations = "/user-config.xml")

})

public class BaseTests {}

@ContextHierarchy(

@ContextConfiguration(name = "child", locations = "/order-config.xml")

)

public class ExtendedTests extends BaseTests {}

Class hierarchy with overridden context hierarchy configuration

In contrast to the previous example, this example demonstrates how to override the configuration for a given named level in a context hierarchy by setting the inheritLocations flag in @ContextConfiguration to false. Consequently, the application context for ExtendedTests will be loaded only from "/test-user-config.xml" and will have its parent set to the context loaded from "/app-config.xml".

@RunWith(SpringRunner.class)

@ContextHierarchy({

@ContextConfiguration(name = "parent", locations = "/app-config.xml"),

@ContextConfiguration(name = "child", locations = "/user-config.xml")

})

public class BaseTests {}

@ContextHierarchy(

@ContextConfiguration(

name = "child",

locations = "/test-user-config.xml",

inheritLocations = false

))

public class ExtendedTests extends BaseTests {}

Dirtying a context within a context hierarchy

If @DirtiesContext is used in a test whose context is configured as part of a context hierarchy, the hierarchyMode flag can be used to control how the context cache is cleared. For further details consult the discussion of @DirtiesContext in Spring Testing Annotations and the @DirtiesContext javadocs.

3.5.5. Dependency injection of test fixtures

When you use the DependencyInjectionTestExecutionListener — which is configured by default — the dependencies of your test instances are injected from beans in the application context that you configured with @ContextConfiguration. You may use setter injection, field injection, or both, depending on which annotations you choose and whether you place them on setter methods or fields. For consistency with the annotation support introduced in Spring 2.5 and 3.0, you can use Spring’s @Autowired annotation or the @Inject annotation from JSR 330.

The TestContext framework does not instrument the manner in which a test instance is instantiated. Thus the use of @Autowired or @Inject for constructors has no effect for test classes.

Because @Autowired is used to perform autowiring by type , if you have multiple bean definitions of the same type, you cannot rely on this approach for those particular beans. In that case, you can use @Autowired in conjunction with @Qualifier. As of Spring 3.0 you may also choose to use @Inject in conjunction with @Named. Alternatively, if your test class has access to its ApplicationContext, you can perform an explicit lookup by using (for example) a call to applicationContext.getBean("titleRepository").

If you do not want dependency injection applied to your test instances, simply do not annotate fields or setter methods with @Autowired or @Inject. Alternatively, you can disable dependency injection altogether by explicitly configuring your class with @TestExecutionListeners and omitting DependencyInjectionTestExecutionListener.class from the list of listeners.

Consider the scenario of testing a HibernateTitleRepository class, as outlined in the Goals section. The next two code listings demonstrate the use of @Autowired on fields and setter methods. The application context configuration is presented after all sample code listings.

The dependency injection behavior in the following code listings is not specific to JUnit 4. The same DI techniques can be used in conjunction with any testing framework.

The following examples make calls to static assertion methods such as assertNotNull() but without prepending the call with Assert. In such cases, assume that the method was properly imported through an import static declaration that is not shown in the example.

The first code listing shows a JUnit 4 based implementation of the test class that uses @Autowired for field injection.

@RunWith(SpringRunner.class)

// specifies the Spring configuration to load for this test fixture

@ContextConfiguration("repository-config.xml")

public class HibernateTitleRepositoryTests {

// this instance will be dependency injected by type

@Autowired

private HibernateTitleRepository titleRepository;

@Test

public void findById() {

Title title = titleRepository.findById(new Long(10));

assertNotNull(title);

}

}

Alternatively, you can configure the class to use @Autowired for setter injection as seen below.

@RunWith(SpringRunner.class)

// specifies the Spring configuration to load for this test fixture

@ContextConfiguration("repository-config.xml")

public class HibernateTitleRepositoryTests {

// this instance will be dependency injected by type

private HibernateTitleRepository titleRepository;

@Autowired

public void setTitleRepository(HibernateTitleRepository titleRepository) {

this.titleRepository = titleRepository;

}

@Test

public void findById() {

Title title = titleRepository.findById(new Long(10));

assertNotNull(title);

}

}

The preceding code listings use the same XML context file referenced by the @ContextConfiguration annotation (that is, repository-config.xml), which looks like this:

<?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">

<!-- this bean will be injected into the HibernateTitleRepositoryTests class -->

<bean id="titleRepository" class="com.foo.repository.hibernate.HibernateTitleRepository">

<property name="sessionFactory" ref="sessionFactory"/>

</bean>

<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

<!-- configuration elided for brevity -->

</bean>

</beans>

If you are extending from a Spring-provided test base class that happens to use @Autowired on one of its setter methods, you might have multiple beans of the affected type defined in your application context: for example, multiple DataSource beans. In such a case, you can override the setter method and use the @Qualifier annotation to indicate a specific target bean as follows, but make sure to delegate to the overridden method in the superclass as well.

// ...

@Autowired

@Override

public void setDataSource(@Qualifier("myDataSource") DataSource dataSource) {

super.setDataSource(dataSource);

}

// ...

The specified qualifier value indicates the specific DataSource bean to inject, narrowing the set of type matches to a specific bean. Its value is matched against <qualifier> declarations within the corresponding <bean> definitions. The bean name is used as a fallback qualifier value, so you may effectively also point to a specific bean by name there (as shown above, assuming that "myDataSource" is the bean id).

3.5.6. Testing request and session scoped beans

Request and session scoped beans have been supported by Spring since the early years, and since Spring 3.2 it’s a breeze to test your request-scoped and session-scoped beans by following these steps.

Ensure that a WebApplicationContext is loaded for your test by annotating your test class with @WebAppConfiguration.

Inject the mock request or session into your test instance and prepare your test fixture as appropriate.

Invoke your web component that you retrieved from the configured WebApplicationContext (i.e., via dependency injection).

Perform assertions against the mocks.

The following code snippet displays the XML configuration for a login use case. Note that the userService bean has a dependency on a request-scoped loginAction bean. Also, the LoginAction is instantiated using SpEL expressions that retrieve the username and password from the current HTTP request. In our test, we will want to configure these request parameters via the mock managed by the TestContext framework.

Request-scoped bean configuration

<beans>

<bean id="userService" class="com.example.SimpleUserService"

c:loginAction-ref="loginAction"/>

<bean id="loginAction" class="com.example.LoginAction"

c:username="#{request.getParameter('user')}"

c:password="#{request.getParameter('pswd')}"

scope="request">

<aop:scoped-proxy/>

</bean>

</beans>

In RequestScopedBeanTests we inject both the UserService (i.e., the subject under test) and the MockHttpServletRequest into our test instance. Within our requestScope() test method we set up our test fixture by setting request parameters in the provided MockHttpServletRequest. When the loginUser() method is invoked on our userService we are assured that the user service has access to the request-scoped loginAction for the current MockHttpServletRequest (i.e., the one we just set parameters in). We can then perform assertions against the results based on the known inputs for the username and password.

Request-scoped bean test

@RunWith(SpringRunner.class)

@ContextConfiguration

@WebAppConfiguration

public class RequestScopedBeanTests {

@Autowired UserService userService;

@Autowired MockHttpServletRequest request;

@Test

public void requestScope() {

request.setParameter("user", "enigma");

request.setParameter("pswd", "$pr!ng");

LoginResults results = userService.loginUser();

// assert results

}

}

The following code snippet is similar to the one we saw above for a request-scoped bean; however, this time the userService bean has a dependency on a session-scoped userPreferences bean. Note that the UserPreferences bean is instantiated using a SpEL expression that retrieves the theme from the current HTTP session. In our test, we will need to configure a theme in the mock session managed by the TestContext framework.

Session-scoped bean configuration

<beans>

<bean id="userService" class="com.example.SimpleUserService"

c:userPreferences-ref="userPreferences" />

<bean id="userPreferences" class="com.example.UserPreferences"

c:theme="#{session.getAttribute('theme')}"

scope="session">

<aop:scoped-proxy/>

</bean>

</beans>

In SessionScopedBeanTests we inject the UserService and the MockHttpSession into our test instance. Within our sessionScope() test method we set up our test fixture by setting the expected "theme" attribute in the provided MockHttpSession. When the processUserPreferences() method is invoked on our userService we are assured that the user service has access to the session-scoped userPreferences for the current MockHttpSession, and we can perform assertions against the results based on the configured theme.

Session-scoped bean test

@RunWith(SpringRunner.class)

@ContextConfiguration

@WebAppConfiguration

public class SessionScopedBeanTests {

@Autowired UserService userService;

@Autowired MockHttpSession session;

@Test

public void sessionScope() throws Exception {

session.setAttribute("theme", "blue");

Results results = userService.processUserPreferences();

// assert results

}

}

3.5.7. Transaction management

In the TestContext framework, transactions are managed by the TransactionalTestExecutionListener which is configured by default, even if you do not explicitly declare @TestExecutionListeners on your test class. To enable support for transactions, however, you must configure a PlatformTransactionManager bean in the ApplicationContext that is loaded via @ContextConfiguration semantics (further details are provided below). In addition, you must declare Spring’s @Transactional annotation either at the class or method level for your tests.

Test-managed transactions

Test-managed transactions are transactions that are managed declaratively via the TransactionalTestExecutionListener or programmatically via TestTransaction (see below). Such transactions should not be confused with Spring-managed transactions (i.e., those managed directly by Spring within the ApplicationContext loaded for tests) or application-managed transactions (i.e., those managed programmatically within application code that is invoked via tests). Spring-managed and application-managed transactions will typically participate in test-managed transactions; however, caution should be taken if Spring-managed or application-managed transactions are configured with any propagation type other than REQUIRED or SUPPORTS (see the discussion on transaction propagation for details).

Enabling and disabling transactions

Annotating a test method with @Transactional causes the test to be run within a transaction that will, by default, be automatically rolled back after completion of the test. If a test class is annotated with @Transactional, each test method within that class hierarchy will be run within a transaction. Test methods that are not annotated with @Transactional (at the class or method level) will not be run within a transaction. Furthermore, tests that are annotated with @Transactional but have the propagation type set to NOT\_SUPPORTED will not be run within a transaction.

Note that AbstractTransactionalJUnit4SpringContextTests and AbstractTransactionalTestNGSpringContextTests are preconfigured for transactional support at the class level.

The following example demonstrates a common scenario for writing an integration test for a Hibernate-based UserRepository. As explained in Transaction rollback and commit behavior, there is no need to clean up the database after the createUser() method is executed since any changes made to the database will be automatically rolled back by the TransactionalTestExecutionListener. See PetClinic Example for an additional example.

@RunWith(SpringRunner.class)

@ContextConfiguration(classes = TestConfig.class)

@Transactional

public class HibernateUserRepositoryTests {

@Autowired

HibernateUserRepository repository;

@Autowired

SessionFactory sessionFactory;

JdbcTemplate jdbcTemplate;

@Autowired

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

@Test

public void createUser() {

// track initial state in test database:

final int count = countRowsInTable("user");

User user = new User(...);

repository.save(user);

// Manual flush is required to avoid false positive in test

sessionFactory.getCurrentSession().flush();

assertNumUsers(count + 1);

}

protected int countRowsInTable(String tableName) {

return JdbcTestUtils.countRowsInTable(this.jdbcTemplate, tableName);

}

protected void assertNumUsers(int expected) {

assertEquals("Number of rows in the [user] table.", expected, countRowsInTable("user"));

}

}

Transaction rollback and commit behavior

By default, test transactions will be automatically rolled back after completion of the test; however, transactional commit and rollback behavior can be configured declaratively via the @Commit and @Rollback annotations. See the corresponding entries in the annotation support section for further details.

Programmatic transaction management

Since Spring Framework 4.1, it is possible to interact with test-managed transactions programmatically via the static methods in TestTransaction. For example, TestTransaction may be used within test methods, before methods, and after methods to start or end the current test-managed transaction or to configure the current test-managed transaction for rollback or commit. Support for TestTransaction is automatically available whenever the TransactionalTestExecutionListener is enabled.

The following example demonstrates some of the features of TestTransaction. Consult the javadocs for TestTransaction for further details.

@ContextConfiguration(classes = TestConfig.class)

public class ProgrammaticTransactionManagementTests extends

AbstractTransactionalJUnit4SpringContextTests {

@Test

public void transactionalTest() {

// assert initial state in test database:

assertNumUsers(2);

deleteFromTables("user");

// changes to the database will be committed!

TestTransaction.flagForCommit();

TestTransaction.end();

assertFalse(TestTransaction.isActive());

assertNumUsers(0);

TestTransaction.start();

// perform other actions against the database that will

// be automatically rolled back after the test completes...

}

protected void assertNumUsers(int expected) {

assertEquals("Number of rows in the [user] table.", expected, countRowsInTable("user"));

}

}

Executing code outside of a transaction

Occasionally you need to execute certain code before or after a transactional test method but outside the transactional context — for example, to verify the initial database state prior to execution of your test or to verify expected transactional commit behavior after test execution (if the test was configured to commit the transaction). TransactionalTestExecutionListener supports the @BeforeTransaction and @AfterTransaction annotations exactly for such scenarios. Simply annotate any void method in a test class or any void default method in a test interface with one of these annotations, and the TransactionalTestExecutionListener ensures that your before transaction method or after transaction method is executed at the appropriate time.

Any before methods (such as methods annotated with JUnit Jupiter’s @BeforeEach) and any after methods (such as methods annotated with JUnit Jupiter’s @AfterEach) are executed within a transaction. In addition, methods annotated with @BeforeTransaction or @AfterTransaction are naturally not executed for test methods that are not configured to run within a transaction.

Configuring a transaction manager

TransactionalTestExecutionListener expects a PlatformTransactionManager bean to be defined in the Spring ApplicationContext for the test. In case there are multiple instances of PlatformTransactionManager within the test’s ApplicationContext, a qualifier may be declared via @Transactional("myTxMgr") or @Transactional(transactionManager = "myTxMgr"), or TransactionManagementConfigurer can be implemented by an @Configuration class. Consult the javadocs for TestContextTransactionUtils.retrieveTransactionManager() for details on the algorithm used to look up a transaction manager in the test’s ApplicationContext.

Demonstration of all transaction-related annotations

The following JUnit 4 based example displays a fictitious integration testing scenario highlighting all transaction-related annotations. The example is not intended to demonstrate best practices but rather to demonstrate how these annotations can be used. Consult the annotation support section for further information and configuration examples. Transaction management for @Sql contains an additional example using @Sql for declarative SQL script execution with default transaction rollback semantics.

@RunWith(SpringRunner.class)

@ContextConfiguration

@Transactional(transactionManager = "txMgr")

@Commit

public class FictitiousTransactionalTest {

@BeforeTransaction

void verifyInitialDatabaseState() {

// logic to verify the initial state before a transaction is started

}

@Before

public void setUpTestDataWithinTransaction() {

// set up test data within the transaction

}

@Test

// overrides the class-level @Commit setting

@Rollback

public void modifyDatabaseWithinTransaction() {

// logic which uses the test data and modifies database state

}

@After

public void tearDownWithinTransaction() {

// execute "tear down" logic within the transaction

}

@AfterTransaction

void verifyFinalDatabaseState() {

// logic to verify the final state after transaction has rolled back

}

}

Avoid false positives when testing ORM code

When you test application code that manipulates the state of a Hibernate session or JPA persistence context, make sure to flush the underlying unit of work within test methods that execute that code. Failing to flush the underlying unit of work can produce false positives: your test may pass, but the same code throws an exception in a live, production environment. In the following Hibernate-based example test case, one method demonstrates a false positive, and the other method correctly exposes the results of flushing the session. Note that this applies to any ORM frameworks that maintain an in-memory unit of work.

// ...

@Autowired

SessionFactory sessionFactory;

@Transactional

@Test // no expected exception!

public void falsePositive() {

updateEntityInHibernateSession();

// False positive: an exception will be thrown once the Hibernate

// Session is finally flushed (i.e., in production code)

}

@Transactional

@Test(expected = ...)

public void updateWithSessionFlush() {

updateEntityInHibernateSession();

// Manual flush is required to avoid false positive in test

sessionFactory.getCurrentSession().flush();

}

// ...

Or for JPA:

// ...

@PersistenceContext

EntityManager entityManager;

@Transactional

@Test // no expected exception!

public void falsePositive() {

updateEntityInJpaPersistenceContext();

// False positive: an exception will be thrown once the JPA

// EntityManager is finally flushed (i.e., in production code)

}

@Transactional

@Test(expected = ...)

public void updateWithEntityManagerFlush() {

updateEntityInJpaPersistenceContext();

// Manual flush is required to avoid false positive in test

entityManager.flush();

}

// ...

3.5.8. Executing SQL scripts

When writing integration tests against a relational database, it is often beneficial to execute SQL scripts to modify the database schema or insert test data into tables. The spring-jdbc module provides support for initializing an embedded or existing database by executing SQL scripts when the Spring ApplicationContext is loaded. See Embedded database support and Testing data access logic with an embedded database for details.

Although it is very useful to initialize a database for testing once when the ApplicationContext is loaded, sometimes it is essential to be able to modify the database during integration tests. The following sections explain how to execute SQL scripts programmatically and declaratively during integration tests.

Executing SQL scripts programmatically

Spring provides the following options for executing SQL scripts programmatically within integration test methods.

org.springframework.jdbc.datasource.init.ScriptUtils

org.springframework.jdbc.datasource.init.ResourceDatabasePopulator

org.springframework.test.context.junit4.AbstractTransactionalJUnit4SpringContextTests

org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTests

ScriptUtils provides a collection of static utility methods for working with SQL scripts and is mainly intended for internal use within the framework. However, if you require full control over how SQL scripts are parsed and executed, ScriptUtils may suit your needs better than some of the other alternatives described below. Consult the javadocs for individual methods in ScriptUtils for further details.

ResourceDatabasePopulator provides a simple object-based API for programmatically populating, initializing, or cleaning up a database using SQL scripts defined in external resources. ResourceDatabasePopulator provides options for configuring the character encoding, statement separator, comment delimiters, and error handling flags used when parsing and executing the scripts, and each of the configuration options has a reasonable default value. Consult the javadocs for details on default values. To execute the scripts configured in a ResourceDatabasePopulator, you can invoke either the populate(Connection) method to execute the populator against a java.sql.Connection or the execute(DataSource) method to execute the populator against a javax.sql.DataSource. The following example specifies SQL scripts for a test schema and test data, sets the statement separator to "@@", and then executes the scripts against a DataSource.

@Test

public void databaseTest {

ResourceDatabasePopulator populator = new ResourceDatabasePopulator();

populator.addScripts(

new ClassPathResource("test-schema.sql"),

new ClassPathResource("test-data.sql"));

populator.setSeparator("@@");

populator.execute(this.dataSource);

// execute code that uses the test schema and data

}

Note that ResourceDatabasePopulator internally delegates to ScriptUtils for parsing and executing SQL scripts. Similarly, the executeSqlScript(..) methods in AbstractTransactionalJUnit4SpringContextTests and AbstractTransactionalTestNGSpringContextTests internally use a ResourceDatabasePopulator for executing SQL scripts. Consult the javadocs for the various executeSqlScript(..) methods for further details.

Executing SQL scripts declaratively with @Sql

In addition to the aforementioned mechanisms for executing SQL scripts programmatically, SQL scripts can also be configured declaratively in the Spring TestContext Framework. Specifically, the @Sql annotation can be declared on a test class or test method to configure the resource paths to SQL scripts that should be executed against a given database either before or after an integration test method. Note that method-level declarations override class-level declarations and that support for @Sql is provided by the SqlScriptsTestExecutionListener which is enabled by default.

Path resource semantics

Each path will be interpreted as a Spring Resource. A plain path — for example, "schema.sql" — will be treated as a classpath resource that is relative to the package in which the test class is defined. A path starting with a slash will be treated as an absolute classpath resource, for example: "/org/example/schema.sql". A path which references a URL (e.g., a path prefixed with classpath:, file:, http:, etc.) will be loaded using the specified resource protocol.

The following example demonstrates how to use @Sql at the class level and at the method level within a JUnit Jupiter based integration test class.

@SpringJUnitConfig

@Sql("/test-schema.sql")

class DatabaseTests {

@Test

void emptySchemaTest {

// execute code that uses the test schema without any test data

}

@Test

@Sql({"/test-schema.sql", "/test-user-data.sql"})

void userTest {

// execute code that uses the test schema and test data

}

}

Default script detection

If no SQL scripts are specified, an attempt will be made to detect a default script depending on where @Sql is declared. If a default cannot be detected, an IllegalStateException will be thrown.

class-level declaration: if the annotated test class is com.example.MyTest, the corresponding default script is "classpath:com/example/MyTest.sql".

method-level declaration: if the annotated test method is named testMethod() and is defined in the class com.example.MyTest, the corresponding default script is "classpath:com/example/MyTest.testMethod.sql".

Declaring multiple @Sql sets

If multiple sets of SQL scripts need to be configured for a given test class or test method but with different syntax configuration, different error handling rules, or different execution phases per set, it is possible to declare multiple instances of @Sql. With Java 8, @Sql can be used as a repeatable annotation. Otherwise, the @SqlGroup annotation can be used as an explicit container for declaring multiple instances of @Sql.

The following example demonstrates the use of @Sql as a repeatable annotation using Java 8. In this scenario the test-schema.sql script uses a different syntax for single-line comments.

@Test

@Sql(scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "`"))

@Sql("/test-user-data.sql")

public void userTest {

// execute code that uses the test schema and test data

}

The following example is identical to the above except that the @Sql declarations are grouped together within @SqlGroup for compatibility with Java 6 and Java 7.

@Test

@SqlGroup({

@Sql(scripts = "/test-schema.sql", config = @SqlConfig(commentPrefix = "`")),

@Sql("/test-user-data.sql")

)}

public void userTest {

// execute code that uses the test schema and test data

}

Script execution phases

By default, SQL scripts will be executed before the corresponding test method. However, if a particular set of scripts needs to be executed after the test method — for example, to clean up database state — the executionPhase attribute in @Sql can be used as seen in the following example. Note that ISOLATED and AFTER\_TEST\_METHOD are statically imported from Sql.TransactionMode and Sql.ExecutionPhase respectively.

@Test

@Sql(

scripts = "create-test-data.sql",

config = @SqlConfig(transactionMode = ISOLATED)

)

@Sql(

scripts = "delete-test-data.sql",

config = @SqlConfig(transactionMode = ISOLATED),

executionPhase = AFTER\_TEST\_METHOD

)

public void userTest {

// execute code that needs the test data to be committed

// to the database outside of the test's transaction

}

Script configuration with @SqlConfig

Configuration for script parsing and error handling can be configured via the @SqlConfig annotation. When declared as a class-level annotation on an integration test class, @SqlConfig serves as global configuration for all SQL scripts within the test class hierarchy. When declared directly via the config attribute of the @Sql annotation, @SqlConfig serves as local configuration for the SQL scripts declared within the enclosing @Sql annotation. Every attribute in @SqlConfig has an implicit default value which is documented in the javadocs of the corresponding attribute. Due to the rules defined for annotation attributes in the Java Language Specification, it is unfortunately not possible to assign a value of null to an annotation attribute. Thus, in order to support overrides of inherited global configuration, @SqlConfig attributes have an explicit default value of either "" for Strings or DEFAULT for Enums. This approach allows local declarations of @SqlConfig to selectively override individual attributes from global declarations of @SqlConfig by providing a value other than "" or DEFAULT. Global @SqlConfig attributes are inherited whenever local @SqlConfig attributes do not supply an explicit value other than "" or DEFAULT. Explicit local configuration therefore overrides global configuration.

The configuration options provided by @Sql and @SqlConfig are equivalent to those supported by ScriptUtils and ResourceDatabasePopulator but are a superset of those provided by the <jdbc:initialize-database/> XML namespace element. Consult the javadocs of individual attributes in @Sql and @SqlConfig for details.

Transaction management for @Sql

By default, the SqlScriptsTestExecutionListener will infer the desired transaction semantics for scripts configured via @Sql. Specifically, SQL scripts will be executed without a transaction, within an existing Spring-managed transaction — for example, a transaction managed by the TransactionalTestExecutionListener for a test annotated with @Transactional — or within an isolated transaction, depending on the configured value of the transactionMode attribute in @SqlConfig and the presence of a PlatformTransactionManager in the test’s ApplicationContext. As a bare minimum however, a javax.sql.DataSource must be present in the test’s ApplicationContext.

If the algorithms used by SqlScriptsTestExecutionListener to detect a DataSource and PlatformTransactionManager and infer the transaction semantics do not suit your needs, you may specify explicit names via the dataSource and transactionManager attributes of @SqlConfig. Furthermore, the transaction propagation behavior can be controlled via the transactionMode attribute of @SqlConfig — for example, if scripts should be executed in an isolated transaction. Although a thorough discussion of all supported options for transaction management with @Sql is beyond the scope of this reference manual, the javadocs for @SqlConfig and SqlScriptsTestExecutionListener provide detailed information, and the following example demonstrates a typical testing scenario using JUnit Jupiter and transactional tests with @Sql. Note that there is no need to clean up the database after the usersTest() method is executed since any changes made to the database (either within the test method or within the /test-data.sql script) will be automatically rolled back by the TransactionalTestExecutionListener (see transaction management for details).

@SpringJUnitConfig(TestDatabaseConfig.class)

@Transactional

class TransactionalSqlScriptsTests {

final JdbcTemplate jdbcTemplate;

@Autowired

TransactionalSqlScriptsTests(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

@Test

@Sql("/test-data.sql")

void usersTest() {

// verify state in test database:

assertNumUsers(2);

// execute code that uses the test data...

}

int countRowsInTable(String tableName) {

return JdbcTestUtils.countRowsInTable(this.jdbcTemplate, tableName);

}

void assertNumUsers(int expected) {

assertEquals(expected, countRowsInTable("user"),

"Number of rows in the [user] table.");

}

}

3.5.9. Parallel test execution

Spring Framework 5.0 introduces basic support for executing tests in parallel within a single JVM when using the Spring TestContext Framework. In general this means that most test classes or test methods can be executed in parallel without any changes to test code or configuration.

For details on how to set up parallel test execution, consult the documentation for your testing framework, build tool, or IDE.

Keep in mind that the introduction of concurrency into your test suite can result in unexpected side effects, strange runtime behavior, and tests that only fail intermittently or seemingly randomly. The Spring Team therefore provides the following general guidelines for when not to execute tests in parallel.

Do not execute tests in parallel if:

Tests make use of Spring’s @DirtiesContext support.

Tests make use of JUnit 4’s @FixMethodOrder support or any testing framework feature that is designed to ensure that test methods execute in a particular order. Note, however, that this does not apply if entire test classes are executed in parallel.

Tests change the state of shared services or systems such as a database, message broker, filesystem, etc. This applies to both in-memory and external systems.

If parallel test execution fails with an exception stating that the ApplicationContext for the current test is no longer active, this typically means that the ApplicationContext was removed from the ContextCache in a different thread.

This may be due to the use of @DirtiesContext or due to automatic eviction from the ContextCache. If @DirtiesContext is the culprit, you will either need to find a way to avoid using @DirtiesContext or exclude such tests from parallel execution. If the maximum size of the ContextCache has been exceeded, you can increase the maximum size of the cache. See the discussion on context caching for details.

Parallel test execution in the Spring TestContext Framework is only possible if the underlying TestContext implementation provides a copy constructor as explained in the javadocs for TestContext. The DefaultTestContext used in Spring provides such a constructor; however, if you use a third-party library that provides a custom TestContext implementation, you will need to verify if it is suitable for parallel test execution.

3.5.10. TestContext Framework support classes

Spring JUnit 4 Runner

The Spring TestContext Framework offers full integration with JUnit 4 through a custom runner (supported on JUnit 4.12 or higher). By annotating test classes with @RunWith(SpringJUnit4ClassRunner.class) or the shorter @RunWith(SpringRunner.class) variant, developers can implement standard JUnit 4 based unit and integration tests and simultaneously reap the benefits of the TestContext framework such as support for loading application contexts, dependency injection of test instances, transactional test method execution, and so on. If you would like to use the Spring TestContext Framework with an alternative runner such as JUnit 4’s Parameterized or third-party runners such as the MockitoJUnitRunner, you may optionally use Spring’s support for JUnit rules instead.

The following code listing displays the minimal requirements for configuring a test class to run with the custom Spring Runner. @TestExecutionListeners is configured with an empty list in order to disable the default listeners, which otherwise would require an ApplicationContext to be configured through @ContextConfiguration.

@RunWith(SpringRunner.class)

@TestExecutionListeners({})

public class SimpleTest {

@Test

public void testMethod() {

// execute test logic...

}

}

Spring JUnit 4 Rules

The org.springframework.test.context.junit4.rules package provides the following JUnit 4 rules (supported on JUnit 4.12 or higher).

SpringClassRule

SpringMethodRule

SpringClassRule is a JUnit TestRule that supports class-level features of the Spring TestContext Framework; whereas, SpringMethodRule is a JUnit MethodRule that supports instance-level and method-level features of the Spring TestContext Framework.

In contrast to the SpringRunner, Spring’s rule-based JUnit support has the advantage that it is independent of any org.junit.runner.Runner implementation and can therefore be combined with existing alternative runners like JUnit 4’s Parameterized or third-party runners such as the MockitoJUnitRunner.

In order to support the full functionality of the TestContext framework, a SpringClassRule must be combined with a SpringMethodRule. The following example demonstrates the proper way to declare these rules in an integration test.

// Optionally specify a non-Spring Runner via @RunWith(...)

@ContextConfiguration

public class IntegrationTest {

@ClassRule

public static final SpringClassRule springClassRule = new SpringClassRule();

@Rule

public final SpringMethodRule springMethodRule = new SpringMethodRule();

@Test

public void testMethod() {

// execute test logic...

}

}

JUnit 4 support classes

The org.springframework.test.context.junit4 package provides the following support classes for JUnit 4 based test cases (supported on JUnit 4.12 or higher).

AbstractJUnit4SpringContextTests

AbstractTransactionalJUnit4SpringContextTests

AbstractJUnit4SpringContextTests is an abstract base test class that integrates the Spring TestContext Framework with explicit ApplicationContext testing support in a JUnit 4 environment. When you extend AbstractJUnit4SpringContextTests, you can access a protected applicationContext instance variable that can be used to perform explicit bean lookups or to test the state of the context as a whole.

AbstractTransactionalJUnit4SpringContextTests is an abstract transactional extension of AbstractJUnit4SpringContextTests that adds some convenience functionality for JDBC access. This class expects a javax.sql.DataSource bean and a PlatformTransactionManager bean to be defined in the ApplicationContext. When you extend AbstractTransactionalJUnit4SpringContextTests you can access a protected jdbcTemplate instance variable that can be used to execute SQL statements to query the database. Such queries can be used to confirm database state both prior to and after execution of database-related application code, and Spring ensures that such queries run in the scope of the same transaction as the application code. When used in conjunction with an ORM tool, be sure to avoid false positives. As mentioned in JDBC Testing Support, AbstractTransactionalJUnit4SpringContextTests also provides convenience methods which delegate to methods in JdbcTestUtils using the aforementioned jdbcTemplate. Furthermore, AbstractTransactionalJUnit4SpringContextTests provides an executeSqlScript(..) method for executing SQL scripts against the configured DataSource.

These classes are a convenience for extension. If you do not want your test classes to be tied to a Spring-specific class hierarchy, you can configure your own custom test classes by using @RunWith(SpringRunner.class) or Spring’s JUnit rules.

SpringExtension for JUnit Jupiter

The Spring TestContext Framework offers full integration with the JUnit Jupiter testing framework introduced in JUnit 5. By annotating test classes with @ExtendWith(SpringExtension.class), developers can implement standard JUnit Jupiter based unit and integration tests and simultaneously reap the benefits of the TestContext framework such as support for loading application contexts, dependency injection of test instances, transactional test method execution, and so on.

Furthermore, thanks to the rich extension API in JUnit Jupiter, Spring is able to provide the following features above and beyond the feature set that Spring supports for JUnit 4 and TestNG.

Dependency injection for test constructors, test methods, and test lifecycle callback methods

See Dependency Injection with the SpringExtension for further details.

Powerful support for conditional test execution based on SpEL expressions, environment variables, system properties, etc.

See the documentation for @EnabledIf and @DisabledIf in Spring JUnit Jupiter Testing Annotations for further details and examples.

Custom composed annotations that combine annotations from Spring and JUnit Jupiter.

See the @TransactionalDevTestConfig and @TransactionalIntegrationTest examples in Meta-Annotation Support for Testing for further details.

The following code listing demonstrates how to configure a test class to use the SpringExtension in conjunction with @ContextConfiguration.

// Instructs JUnit Jupiter to extend the test with Spring support.

@ExtendWith(SpringExtension.class)

// Instructs Spring to load an ApplicationContext from TestConfig.class

@ContextConfiguration(classes = TestConfig.class)

class SimpleTests {

@Test

void testMethod() {

// execute test logic...

}

}

Since annotations in JUnit 5 can also be used as meta-annotations, Spring is able to provide @SpringJUnitConfig and @SpringJUnitWebConfig composed annotations to simplify the configuration of the test ApplicationContext and JUnit Jupiter.

For example, the following example uses @SpringJUnitConfig to reduce the amount of configuration used in the previous example.

// Instructs Spring to register the SpringExtension with JUnit

// Jupiter and load an ApplicationContext from TestConfig.class

@SpringJUnitConfig(TestConfig.class)

class SimpleTests {

@Test

void testMethod() {

// execute test logic...

}

}

Similarly, the following example uses @SpringJUnitWebConfig to create a WebApplicationContext for use with JUnit Jupiter.

// Instructs Spring to register the SpringExtension with JUnit

// Jupiter and load a WebApplicationContext from TestWebConfig.class

@SpringJUnitWebConfig(TestWebConfig.class)

class SimpleWebTests {

@Test

void testMethod() {

// execute test logic...

}

}

See the documentation for @SpringJUnitConfig and @SpringJUnitWebConfig in Spring JUnit Jupiter Testing Annotations for further details.

Dependency Injection with the SpringExtension

The SpringExtension implements the ParameterResolver extension API from JUnit Jupiter which allows Spring to provide dependency injection for test constructors, test methods, and test lifecycle callback methods.

Specifically, the SpringExtension is able to inject dependencies from the test’s ApplicationContext into test constructors and methods annotated with @BeforeAll, @AfterAll, @BeforeEach, @AfterEach, @Test, @RepeatedTest, @ParameterizedTest, etc.

Constructor Injection

If a parameter in a constructor for a JUnit Jupiter test class is of type ApplicationContext (or a sub-type thereof) or is annotated or meta-annotated with @Autowired, @Qualifier, or @Value, Spring will inject the value for that specific parameter with the corresponding bean from the test’s ApplicationContext. A test constructor can also be directly annotated with @Autowired if all of the parameters should be supplied by Spring.

If the constructor for a test class is itself annotated with @Autowired, Spring will assume the responsibility for resolving all parameters in the constructor. Consequently, no other ParameterResolver registered with JUnit Jupiter will be able to resolve parameters for such a constructor.

In the following example, Spring will inject the OrderService bean from the ApplicationContext loaded from TestConfig.class into the OrderServiceIntegrationTests constructor. Note as well that this feature allows test dependencies to be final and therefore immutable.

@SpringJUnitConfig(TestConfig.class)

class OrderServiceIntegrationTests {

private final OrderService orderService;

@Autowired

OrderServiceIntegrationTests(OrderService orderService) {

this.orderService = orderService.

}

// tests that use the injected OrderService

}

Method Injection

If a parameter in a JUnit Jupiter test method or test lifecycle callback method is of type ApplicationContext (or a sub-type thereof) or is annotated or meta-annotated with @Autowired, @Qualifier, or @Value, Spring will inject the value for that specific parameter with the corresponding bean from the test’s ApplicationContext.

In the following example, Spring will inject the OrderService from the ApplicationContext loaded from TestConfig.class into the deleteOrder() test method.

@SpringJUnitConfig(TestConfig.class)

class OrderServiceIntegrationTests {

@Test

void deleteOrder(@Autowired OrderService orderService) {

// use orderService from the test's ApplicationContext

}

}

Due to the robustness of the ParameterResolver support in JUnit Jupiter, it is also possible to have multiple dependencies injected into a single method not only from Spring but also from JUnit Jupiter itself or other third-party extensions.

The following example demonstrates how to have both Spring and JUnit Jupiter inject dependencies into the placeOrderRepeatedly() test method simultaneously. Note that the use of @RepeatedTest from JUnit Jupiter allows the test method to gain access to the RepetitionInfo.

@SpringJUnitConfig(TestConfig.class)

class OrderServiceIntegrationTests {

@RepeatedTest(10)

void placeOrderRepeatedly(RepetitionInfo repetitionInfo,

@Autowired OrderService orderService) {

// use orderService from the test's ApplicationContext

// and repetitionInfo from JUnit Jupiter

}

}

TestNG support classes

The org.springframework.test.context.testng package provides the following support classes for TestNG based test cases.

AbstractTestNGSpringContextTests

AbstractTransactionalTestNGSpringContextTests

AbstractTestNGSpringContextTests is an abstract base test class that integrates the Spring TestContext Framework with explicit ApplicationContext testing support in a TestNG environment. When you extend AbstractTestNGSpringContextTests, you can access a protected applicationContext instance variable that can be used to perform explicit bean lookups or to test the state of the context as a whole.

AbstractTransactionalTestNGSpringContextTests is an abstract transactional extension of AbstractTestNGSpringContextTests that adds some convenience functionality for JDBC access. This class expects a javax.sql.DataSource bean and a PlatformTransactionManager bean to be defined in the ApplicationContext. When you extend AbstractTransactionalTestNGSpringContextTests you can access a protected jdbcTemplate instance variable that can be used to execute SQL statements to query the database. Such queries can be used to confirm database state both prior to and after execution of database-related application code, and Spring ensures that such queries run in the scope of the same transaction as the application code. When used in conjunction with an ORM tool, be sure to avoid false positives. As mentioned in JDBC Testing Support, AbstractTransactionalTestNGSpringContextTests also provides convenience methods which delegate to methods in JdbcTestUtils using the aforementioned jdbcTemplate. Furthermore, AbstractTransactionalTestNGSpringContextTests provides an executeSqlScript(..) method for executing SQL scripts against the configured DataSource.

These classes are a convenience for extension. If you do not want your test classes to be tied to a Spring-specific class hierarchy, you can configure your own custom test classes by using @ContextConfiguration, @TestExecutionListeners, and so on, and by manually instrumenting your test class with a TestContextManager. See the source code of AbstractTestNGSpringContextTests for an example of how to instrument your test class.

3.6. Spring MVC Test Framework

The Spring MVC Test framework provides first class support for testing Spring MVC code using a fluent API that can be used with JUnit, TestNG, or any other testing framework. It’s built on the Servlet API mock objects from the spring-test module and hence does not use a running Servlet container. It uses the DispatcherServlet to provide full Spring MVC runtime behavior and provides support for loading actual Spring configuration with the TestContext framework in addition to a standalone mode in which controllers may be instantiated manually and tested one at a time.

Spring MVC Test also provides client-side support for testing code that uses the RestTemplate. Client-side tests mock the server responses and also do not use a running server.

Spring Boot provides an option to write full, end-to-end integration tests that include a running server. If this is your goal please have a look at the Spring Boot reference page. For more information on the differences between out-of-container and end-to-end integration tests, see Differences between Out-of-Container and End-to-End Integration Tests.

3.6.1. Server-Side Tests

It’s easy to write a plain unit test for a Spring MVC controller using JUnit or TestNG: simply instantiate the controller, inject it with mocked or stubbed dependencies, and call its methods passing MockHttpServletRequest, MockHttpServletResponse, etc., as necessary. However, when writing such a unit test, much remains untested: for example, request mappings, data binding, type conversion, validation, and much more. Furthermore, other controller methods such as @InitBinder, @ModelAttribute, and @ExceptionHandler may also be invoked as part of the request processing lifecycle.

The goal of Spring MVC Test is to provide an effective way for testing controllers by performing requests and generating responses through the actual DispatcherServlet.

Spring MVC Test builds on the familiar "mock" implementations of the Servlet API available in the spring-test module. This allows performing requests and generating responses without the need for running in a Servlet container. For the most part everything should work as it does at runtime with a few notable exceptions as explained in Differences between Out-of-Container and End-to-End Integration Tests. Here is a JUnit Jupiter based example of using Spring MVC Test:

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;

import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

@SpringJUnitWebConfig(locations = "test-servlet-context.xml")

class ExampleTests {

private MockMvc mockMvc;

@BeforeEach

void setup(WebApplicationContext wac) {

this.mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();

}

@Test

void getAccount() throws Exception {

this.mockMvc.perform(get("/accounts/1")

.accept(MediaType.parseMediaType("application/json;charset=UTF-8")))

.andExpect(status().isOk())

.andExpect(content().contentType("application/json"))

.andExpect(jsonPath("$.name").value("Lee"));

}

}

The above test relies on the WebApplicationContext support of the TestContext framework for loading Spring configuration from an XML configuration file located in the same package as the test class, but Java-based and Groovy-based configuration are also supported. See these sample tests.

The MockMvc instance is used to perform a GET request to "/accounts/1" and verify that the resulting response has status 200, the content type is "application/json", and the response body has a JSON property called "name" with the value "Lee". The jsonPath syntax is supported through the Jayway JsonPath project. There are lots of other options for verifying the result of the performed request that will be discussed below.

Static Imports

The fluent API in the example above requires a few static imports such as MockMvcRequestBuilders.\*, MockMvcResultMatchers.\*, and MockMvcBuilders.\*. An easy way to find these classes is to search for types matching "MockMvc\*". If using Eclipse, be sure to add them as "favorite static members" in the Eclipse preferences under Java → Editor → Content Assist → Favorites. That will allow use of content assist after typing the first character of the static method name. Other IDEs (e.g. IntelliJ) may not require any additional configuration. Just check the support for code completion on static members.

Setup Choices

There are two main options for creating an instance of MockMvc. The first is to load Spring MVC configuration through the TestContext framework, which loads the Spring configuration and injects a WebApplicationContext into the test to use to build a MockMvc instance:

@RunWith(SpringRunner.class)

@WebAppConfiguration

@ContextConfiguration("my-servlet-context.xml")

public class MyWebTests {

@Autowired

private WebApplicationContext wac;

private MockMvc mockMvc;

@Before

public void setup() {

this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();

}

// ...

}

The second is to simply create a controller instance manually without loading Spring configuration. Instead basic default configuration, roughly comparable to that of the MVC JavaConfig or the MVC namespace, is automatically created and can be customized to a degree:

public class MyWebTests {

private MockMvc mockMvc;

@Before

public void setup() {

this.mockMvc = MockMvcBuilders.standaloneSetup(new AccountController()).build();

}

// ...

}

Which setup option should you use?

The "webAppContextSetup" loads your actual Spring MVC configuration resulting in a more complete integration test. Since the TestContext framework caches the loaded Spring configuration, it helps keep tests running fast, even as you introduce more tests in your test suite. Furthermore, you can inject mock services into controllers through Spring configuration in order to remain focused on testing the web layer. Here is an example of declaring a mock service with Mockito:

<bean id="accountService" class="org.mockito.Mockito" factory-method="mock">

<constructor-arg value="org.example.AccountService"/>

</bean>

You can then inject the mock service into the test in order set up and verify expectations:

@RunWith(SpringRunner.class)

@WebAppConfiguration

@ContextConfiguration("test-servlet-context.xml")

public class AccountTests {

@Autowired

private WebApplicationContext wac;

private MockMvc mockMvc;

@Autowired

private AccountService accountService;

// ...

}

The "standaloneSetup" on the other hand is a little closer to a unit test. It tests one controller at a time: the controller can be injected with mock dependencies manually, and it doesn’t involve loading Spring configuration. Such tests are more focused on style and make it easier to see which controller is being tested, whether any specific Spring MVC configuration is required to work, and so on. The "standaloneSetup" is also a very convenient way to write ad-hoc tests to verify specific behavior or to debug an issue.

Just like with any "integration vs. unit testing" debate, there is no right or wrong answer. However, using the "standaloneSetup" does imply the need for additional "webAppContextSetup" tests in order to verify your Spring MVC configuration. Alternatively, you may choose to write all tests with "webAppContextSetup" in order to always test against your actual Spring MVC configuration.

Setup Features

No matter which MockMvc builder you use all MockMvcBuilder implementations provide some common and very useful features. For example you can declare an Accept header for all requests and expect a status of 200 as well as a Content-Type header in all responses as follows:

// static import of MockMvcBuilders.standaloneSetup

MockMvc mockMvc = standaloneSetup(new MusicController())

.defaultRequest(get("/").accept(MediaType.APPLICATION\_JSON))

.alwaysExpect(status().isOk())

.alwaysExpect(content().contentType("application/json;charset=UTF-8"))

.build();

In addition 3rd party frameworks (and applications) may pre-package setup instructions like the ones through a MockMvcConfigurer. The Spring Framework has one such built-in implementation that helps to save and re-use the HTTP session across requests. It can be used as follows:

// static import of SharedHttpSessionConfigurer.sharedHttpSession

MockMvc mockMvc = MockMvcBuilders.standaloneSetup(new TestController())

.apply(sharedHttpSession())

.build();

// Use mockMvc to perform requests...

See ConfigurableMockMvcBuilder for a list of all MockMvc builder features or use the IDE to explore the available options.

Performing Requests

It’s easy to perform requests using any HTTP method:

mockMvc.perform(post("/hotels/{id}", 42).accept(MediaType.APPLICATION\_JSON));

You can also perform file upload requests that internally use MockMultipartHttpServletRequest so that there is no actual parsing of a multipart request but rather you have to set it up:

mockMvc.perform(multipart("/doc").file("a1", "ABC".getBytes("UTF-8")));

You can specify query parameters in URI template style:

mockMvc.perform(get("/hotels?foo={foo}", "bar"));

Or you can add Servlet request parameters representing either query of form parameters:

mockMvc.perform(get("/hotels").param("foo", "bar"));

If application code relies on Servlet request parameters and doesn’t check the query string explicitly (as is most often the case) then it doesn’t matter which option you use. Keep in mind however that query params provided with the URI template will be decoded while request parameters provided through the param(…​) method are expected to already be decoded.

In most cases it’s preferable to leave out the context path and the Servlet path from the request URI. If you must test with the full request URI, be sure to set the contextPath and servletPath accordingly so that request mappings will work:

mockMvc.perform(get("/app/main/hotels/{id}").contextPath("/app").servletPath("/main"))

Looking at the above example, it would be cumbersome to set the contextPath and servletPath with every performed request. Instead you can set up default request properties:

public class MyWebTests {

private MockMvc mockMvc;

@Before

public void setup() {

mockMvc = standaloneSetup(new AccountController())

.defaultRequest(get("/")

.contextPath("/app").servletPath("/main")

.accept(MediaType.APPLICATION\_JSON)).build();

}

The above properties will affect every request performed through the MockMvc instance. If the same property is also specified on a given request, it overrides the default value. That is why the HTTP method and URI in the default request don’t matter since they must be specified on every request.

Defining Expectations

Expectations can be defined by appending one or more .andExpect(..) calls after performing a request:

mockMvc.perform(get("/accounts/1")).andExpect(status().isOk());

MockMvcResultMatchers.\* provides a number of expectations, some of which are further nested with more detailed expectations.

Expectations fall in two general categories. The first category of assertions verifies properties of the response: for example, the response status, headers, and content. These are the most important results to assert.

The second category of assertions goes beyond the response. These assertions allow one to inspect Spring MVC specific aspects such as which controller method processed the request, whether an exception was raised and handled, what the content of the model is, what view was selected, what flash attributes were added, and so on. They also allow one to inspect Servlet specific aspects such as request and session attributes.

The following test asserts that binding or validation failed:

mockMvc.perform(post("/persons"))

.andExpect(status().isOk())

.andExpect(model().attributeHasErrors("person"));

Many times when writing tests, it’s useful to dump the results of the performed request. This can be done as follows, where print() is a static import from MockMvcResultHandlers:

mockMvc.perform(post("/persons"))

.andDo(print())

.andExpect(status().isOk())

.andExpect(model().attributeHasErrors("person"));

As long as request processing does not cause an unhandled exception, the print() method will print all the available result data to System.out. Spring Framework 4.2 introduced a log() method and two additional variants of the print() method, one that accepts an OutputStream and one that accepts a Writer. For example, invoking print(System.err) will print the result data to System.err; while invoking print(myWriter) will print the result data to a custom writer. If you would like to have the result data logged instead of printed, simply invoke the log() method which will log the result data as a single DEBUG message under the org.springframework.test.web.servlet.result logging category.

In some cases, you may want to get direct access to the result and verify something that cannot be verified otherwise. This can be achieved by appending .andReturn() after all other expectations:

MvcResult mvcResult = mockMvc.perform(post("/persons")).andExpect(status().isOk()).andReturn();

// ...

If all tests repeat the same expectations you can set up common expectations once when building the MockMvc instance:

standaloneSetup(new SimpleController())

.alwaysExpect(status().isOk())

.alwaysExpect(content().contentType("application/json;charset=UTF-8"))

.build()

Note that common expectations are always applied and cannot be overridden without creating a separate MockMvc instance.

When JSON response content contains hypermedia links created with Spring HATEOAS, the resulting links can be verified using JsonPath expressions:

mockMvc.perform(get("/people").accept(MediaType.APPLICATION\_JSON))

.andExpect(jsonPath("$.links[?(@.rel == 'self')].href").value("http://localhost:8080/people"));

When XML response content contains hypermedia links created with Spring HATEOAS, the resulting links can be verified using XPath expressions:

Map<String, String> ns = Collections.singletonMap("ns", "http://www.w3.org/2005/Atom");

mockMvc.perform(get("/handle").accept(MediaType.APPLICATION\_XML))

.andExpect(xpath("/person/ns:link[@rel='self']/@href", ns).string("http://localhost:8080/people"));

Filter Registrations

When setting up a MockMvc instance, you can register one or more Servlet Filter instances:

mockMvc = standaloneSetup(new PersonController()).addFilters(new CharacterEncodingFilter()).build();

Registered filters will be invoked through via the MockFilterChain from spring-test, and the last filter will delegate to the DispatcherServlet.

Differences between Out-of-Container and End-to-End Integration Tests

As mentioned earlier Spring MVC Test is built on the Servlet API mock objects from the spring-test module and does not use a running Servlet container. Therefore there are some important differences compared to full end-to-end integration tests with an actual client and server running.

The easiest way to think about this is starting with a blank MockHttpServletRequest. Whatever you add to it is what the request will be. Things that may catch you by surprise are that there is no context path by default, no jsessionid cookie, no forwarding, error, or async dispatches, and therefore no actual JSP rendering. Instead, "forwarded" and "redirected" URLs are saved in the MockHttpServletResponse and can be asserted with expectations.

This means if you are using JSPs you can verify the JSP page to which the request was forwarded, but there won’t be any HTML rendered. In other words, the JSP will not be invoked. Note however that all other rendering technologies which don’t rely on forwarding such as Thymeleaf and Freemarker will render HTML to the response body as expected. The same is true for rendering JSON, XML, and other formats via @ResponseBody methods.

Alternatively you may consider the full end-to-end integration testing support from Spring Boot via @WebIntegrationTest. See the Spring Boot reference.

There are pros and cons for each approach. The options provided in Spring MVC Test are different stops on the scale from classic unit testing to full integration testing. To be certain, none of the options in Spring MVC Test fall under the category of classic unit testing, but they are a little closer to it. For example, you can isolate the web layer by injecting mocked services into controllers, in which case you’re testing the web layer only through the DispatcherServlet but with actual Spring configuration, just like you might test the data access layer in isolation from the layers above. Or you can use the standalone setup focusing on one controller at a time and manually providing the configuration required to make it work.

Another important distinction when using Spring MVC Test is that conceptually such tests are on the inside of the server-side so you can check what handler was used, if an exception was handled with a HandlerExceptionResolver, what the content of the model is, what binding errors there were, etc. That means it’s easier to write expectations since the server is not a black box as it is when testing it through an actual HTTP client. This is generally an advantage of classic unit testing, that it’s easier to write, reason about, and debug but does not replace the need for full integration tests. At the same time it’s important not to lose sight of the fact that the response is the most important thing to check. In short, there is room here for multiple styles and strategies of testing even within the same project.

Further Server-Side Test Examples

The framework’s own tests include many sample tests intended to demonstrate how to use Spring MVC Test. Browse these examples for further ideas. Also the spring-mvc-showcase has full test coverage based on Spring MVC Test.

3.6.2. HtmlUnit Integration

Spring provides integration between MockMvc and HtmlUnit. This simplifies performing end-to-end testing when using HTML based views. This integration enables developers to:

Easily test HTML pages using tools such as HtmlUnit, WebDriver, & Geb without the need to deploy to a Servlet container

Test JavaScript within pages

Optionally test using mock services to speed up testing

Share logic between in-container end-to-end tests and out-of-container integration tests

MockMvc works with templating technologies that do not rely on a Servlet Container (e.g., Thymeleaf, FreeMarker, etc.), but it does not work with JSPs since they rely on the Servlet container.

Why HtmlUnit Integration?

The most obvious question that comes to mind is, "Why do I need this?". The answer is best found by exploring a very basic sample application. Assume you have a Spring MVC web application that supports CRUD operations on a Message object. The application also supports paging through all messages. How would you go about testing it?

With Spring MVC Test, we can easily test if we are able to create a Message.

MockHttpServletRequestBuilder createMessage = post("/messages/")

.param("summary", "Spring Rocks")

.param("text", "In case you didn't know, Spring Rocks!");

mockMvc.perform(createMessage)

.andExpect(status().is3xxRedirection())

.andExpect(redirectedUrl("/messages/123"));

What if we want to test our form view that allows us to create the message? For example, assume our form looks like the following snippet:

<form id="messageForm" action="/messages/" method="post">

<div class="pull-right"><a href="/messages/">Messages</a></div>

<label for="summary">Summary</label>

<input type="text" class="required" id="summary" name="summary" value="" />

<label for="text">Message</label>

<textarea id="text" name="text"></textarea>

<div class="form-actions">

<input type="submit" value="Create" />

</div>

</form>

How do we ensure that our form will produce the correct request to create a new message? A naive attempt would look like this:

mockMvc.perform(get("/messages/form"))

.andExpect(xpath("//input[@name='summary']").exists())

.andExpect(xpath("//textarea[@name='text']").exists());

This test has some obvious drawbacks. If we update our controller to use the parameter message instead of text, our form test would continue to pass even though the HTML form is out of synch with the controller. To resolve this we can combine our two tests.

String summaryParamName = "summary";

String textParamName = "text";

mockMvc.perform(get("/messages/form"))

.andExpect(xpath("//input[@name='" + summaryParamName + "']").exists())

.andExpect(xpath("//textarea[@name='" + textParamName + "']").exists());

MockHttpServletRequestBuilder createMessage = post("/messages/")

.param(summaryParamName, "Spring Rocks")

.param(textParamName, "In case you didn't know, Spring Rocks!");

mockMvc.perform(createMessage)

.andExpect(status().is3xxRedirection())

.andExpect(redirectedUrl("/messages/123"));

This would reduce the risk of our test incorrectly passing, but there are still some problems.

What if we have multiple forms on our page? Admittedly we could update our xpath expressions, but they get more complicated the more factors we take into account (Are the fields the correct type? Are the fields enabled? etc.).

Another issue is that we are doing double the work we would expect. We must first verify the view, and then we submit the view with the same parameters we just verified. Ideally this could be done all at once.

Finally, there are some things that we still cannot account for. For example, what if the form has JavaScript validation that we wish to test as well?

The overall problem is that testing a web page does not involve a single interaction. Instead, it is a combination of how the user interacts with a web page and how that web page interacts with other resources. For example, the result of a form view is used as the input to a user for creating a message. In addition, our form view may potentially utilize additional resources which impact the behavior of the page, such as JavaScript validation.

Integration testing to the rescue?

To resolve the issues above we could perform end-to-end integration testing, but this has some obvious drawbacks. Consider testing the view that allows us to page through the messages. We might need the following tests.

Does our page display a notification to the user indicating that no results are available when the messages are empty?

Does our page properly display a single message?

Does our page properly support paging?

To set up these tests, we would need to ensure our database contained the proper messages in it. This leads to a number of additional challenges.

Ensuring the proper messages are in the database can be tedious; consider foreign key constraints.

Testing can become slow since each test would need to ensure that the database is in the correct state.

Since our database needs to be in a specific state, we cannot run tests in parallel.

Performing assertions on things like auto-generated ids, timestamps, etc. can be difficult.

These challenges do not mean that we should abandon end-to-end integration testing altogether. Instead, we can reduce the number of end-to-end integration tests by refactoring our detailed tests to use mock services which will execute much faster, more reliably, and without side effects. We can then implement a small number of true end-to-end integration tests that validate simple workflows to ensure that everything works together properly.

Enter HtmlUnit Integration

So how can we achieve a balance between testing the interactions of our pages and still retain good performance within our test suite? The answer is: "By integrating MockMvc with HtmlUnit."

HtmlUnit Integration Options

There are a number of ways to integrate MockMvc with HtmlUnit.

MockMvc and HtmlUnit: Use this option if you want to use the raw HtmlUnit libraries.

MockMvc and WebDriver: Use this option to ease development and reuse code between integration and end-to-end testing.

MockMvc and Geb: Use this option if you would like to use Groovy for testing, ease development, and reuse code between integration and end-to-end testing.

MockMvc and HtmlUnit

This section describes how to integrate MockMvc and HtmlUnit. Use this option if you want to use the raw HtmlUnit libraries.

MockMvc and HtmlUnit Setup

First, make sure that you have included a test dependency on net.sourceforge.htmlunit:htmlunit. In order to use HtmlUnit with Apache HttpComponents 4.5+, you will need to use HtmlUnit 2.18 or higher.

We can easily create an HtmlUnit WebClient that integrates with MockMvc using the MockMvcWebClientBuilder as follows.

@Autowired

WebApplicationContext context;

WebClient webClient;

@Before

public void setup() {

webClient = MockMvcWebClientBuilder

.webAppContextSetup(context)

.build();

}

This is a simple example of using MockMvcWebClientBuilder. For advanced usage see Advanced MockMvcWebClientBuilder

This will ensure that any URL referencing localhost as the server will be directed to our MockMvc instance without the need for a real HTTP connection. Any other URL will be requested using a network connection as normal. This allows us to easily test the use of CDNs.

MockMvc and HtmlUnit Usage

Now we can use HtmlUnit as we normally would, but without the need to deploy our application to a Servlet container. For example, we can request the view to create a message with the following.

HtmlPage createMsgFormPage = webClient.getPage("http://localhost/messages/form");

The default context path is "". Alternatively, we can specify the context path as illustrated in Advanced MockMvcWebClientBuilder.

Once we have a reference to the HtmlPage, we can then fill out the form and submit it to create a message.

HtmlForm form = createMsgFormPage.getHtmlElementById("messageForm");

HtmlTextInput summaryInput = createMsgFormPage.getHtmlElementById("summary");

summaryInput.setValueAttribute("Spring Rocks");

HtmlTextArea textInput = createMsgFormPage.getHtmlElementById("text");

textInput.setText("In case you didn't know, Spring Rocks!");

HtmlSubmitInput submit = form.getOneHtmlElementByAttribute("input", "type", "submit");

HtmlPage newMessagePage = submit.click();

Finally, we can verify that a new message was created successfully. The following assertions use the AssertJ library.

assertThat(newMessagePage.getUrl().toString()).endsWith("/messages/123");

String id = newMessagePage.getHtmlElementById("id").getTextContent();

assertThat(id).isEqualTo("123");

String summary = newMessagePage.getHtmlElementById("summary").getTextContent();

assertThat(summary).isEqualTo("Spring Rocks");

String text = newMessagePage.getHtmlElementById("text").getTextContent();

assertThat(text).isEqualTo("In case you didn't know, Spring Rocks!");

This improves on our MockMvc test in a number of ways. First we no longer have to explicitly verify our form and then create a request that looks like the form. Instead, we request the form, fill it out, and submit it, thereby significantly reducing the overhead.

Another important factor is that HtmlUnit uses the Mozilla Rhino engine to evaluate JavaScript. This means that we can test the behavior of JavaScript within our pages as well!

Refer to the HtmlUnit documentation for additional information about using HtmlUnit.

Advanced MockMvcWebClientBuilder

In the examples so far, we have used MockMvcWebClientBuilder in the simplest way possible, by building a WebClient based on the WebApplicationContext loaded for us by the Spring TestContext Framework. This approach is repeated here.

@Autowired

WebApplicationContext context;

WebClient webClient;

@Before

public void setup() {

webClient = MockMvcWebClientBuilder

.webAppContextSetup(context)

.build();

}

We can also specify additional configuration options.

WebClient webClient;

@Before

public void setup() {

webClient = MockMvcWebClientBuilder

// demonstrates applying a MockMvcConfigurer (Spring Security)

.webAppContextSetup(context, springSecurity())

// for illustration only - defaults to ""

.contextPath("")

// By default MockMvc is used for localhost only;

// the following will use MockMvc for example.com and example.org as well

.useMockMvcForHosts("example.com","example.org")

.build();

}

As an alternative, we can perform the exact same setup by configuring the MockMvc instance separately and supplying it to the MockMvcWebClientBuilder as follows.

MockMvc mockMvc = MockMvcBuilders

.webAppContextSetup(context)

.apply(springSecurity())

.build();

webClient = MockMvcWebClientBuilder

.mockMvcSetup(mockMvc)

// for illustration only - defaults to ""

.contextPath("")

// By default MockMvc is used for localhost only;

// the following will use MockMvc for example.com and example.org as well

.useMockMvcForHosts("example.com","example.org")

.build();

This is more verbose, but by building the WebClient with a MockMvc instance we have the full power of MockMvc at our fingertips.

For additional information on creating a MockMvc instance refer to Setup Choices.

MockMvc and WebDriver

In the previous sections, we have seen how to use MockMvc in conjunction with the raw HtmlUnit APIs. In this section, we will leverage additional abstractions within the Selenium WebDriver to make things even easier.

Why WebDriver and MockMvc?

We can already use HtmlUnit and MockMvc, so why would we want to use WebDriver? The Selenium WebDriver provides a very elegant API that allows us to easily organize our code. To better understand, let’s explore an example.

Despite being a part of Selenium, WebDriver does not require a Selenium Server to run your tests.

Suppose we need to ensure that a message is created properly. The tests involve finding the HTML form input elements, filling them out, and making various assertions.

This approach results in numerous, separate tests because we want to test error conditions as well. For example, we want to ensure that we get an error if we fill out only part of the form. If we fill out the entire form, the newly created message should be displayed afterwards.

If one of the fields were named "summary", then we might have something like the following repeated in multiple places within our tests.

HtmlTextInput summaryInput = currentPage.getHtmlElementById("summary");

summaryInput.setValueAttribute(summary);

So what happens if we change the id to "smmry"? Doing so would force us to update all of our tests to incorporate this change! Of course, this violates the DRY Principle; so we should ideally extract this code into its own method as follows.

public HtmlPage createMessage(HtmlPage currentPage, String summary, String text) {

setSummary(currentPage, summary);

// ...

}

public void setSummary(HtmlPage currentPage, String summary) {

HtmlTextInput summaryInput = currentPage.getHtmlElementById("summary");

summaryInput.setValueAttribute(summary);

}

This ensures that we do not have to update all of our tests if we change the UI.

We might even take this a step further and place this logic within an Object that represents the HtmlPage we are currently on.

public class CreateMessagePage {

final HtmlPage currentPage;

final HtmlTextInput summaryInput;

final HtmlSubmitInput submit;

public CreateMessagePage(HtmlPage currentPage) {

this.currentPage = currentPage;

this.summaryInput = currentPage.getHtmlElementById("summary");

this.submit = currentPage.getHtmlElementById("submit");

}

public <T> T createMessage(String summary, String text) throws Exception {

setSummary(summary);

HtmlPage result = submit.click();

boolean error = CreateMessagePage.at(result);

return (T) (error ? new CreateMessagePage(result) : new ViewMessagePage(result));

}

public void setSummary(String summary) throws Exception {

summaryInput.setValueAttribute(summary);

}

public static boolean at(HtmlPage page) {

return "Create Message".equals(page.getTitleText());

}

}

Formerly, this pattern is known as the Page Object Pattern. While we can certainly do this with HtmlUnit, WebDriver provides some tools that we will explore in the following sections to make this pattern much easier to implement.

MockMvc and WebDriver Setup

To use Selenium WebDriver with the Spring MVC Test framework, make sure that your project includes a test dependency on org.seleniumhq.selenium:selenium-htmlunit-driver.

We can easily create a Selenium WebDriver that integrates with MockMvc using the MockMvcHtmlUnitDriverBuilder as follows.

@Autowired

WebApplicationContext context;

WebDriver driver;

@Before

public void setup() {

driver = MockMvcHtmlUnitDriverBuilder

.webAppContextSetup(context)

.build();

}

This is a simple example of using MockMvcHtmlUnitDriverBuilder. For more advanced usage, refer to Advanced MockMvcHtmlUnitDriverBuilder

This will ensure that any URL referencing localhost as the server will be directed to our MockMvc instance without the need for a real HTTP connection. Any other URL will be requested using a network connection as normal. This allows us to easily test the use of CDNs.

MockMvc and WebDriver Usage

Now we can use WebDriver as we normally would, but without the need to deploy our application to a Servlet container. For example, we can request the view to create a message with the following.

CreateMessagePage page = CreateMessagePage.to(driver);

We can then fill out the form and submit it to create a message.

ViewMessagePage viewMessagePage =

page.createMessage(ViewMessagePage.class, expectedSummary, expectedText);

This improves on the design of our HtmlUnit test by leveraging the Page Object Pattern. As we mentioned in Why WebDriver and MockMvc?, we can use the Page Object Pattern with HtmlUnit, but it is much easier with WebDriver. Let’s take a look at our new CreateMessagePage implementation.

public class CreateMessagePage

extends AbstractPage {

private WebElement summary;

private WebElement text;

@FindBy(css = "input[type=submit]")

private WebElement submit;

public CreateMessagePage(WebDriver driver) {

super(driver);

}

public <T> T createMessage(Class<T> resultPage, String summary, String details) {

this.summary.sendKeys(summary);

this.text.sendKeys(details);

this.submit.click();

return PageFactory.initElements(driver, resultPage);

}

public static CreateMessagePage to(WebDriver driver) {

driver.get("http://localhost:9990/mail/messages/form");

return PageFactory.initElements(driver, CreateMessagePage.class);

}

}

The first thing you will notice is that CreateMessagePage extends the AbstractPage. We won’t go over the details of AbstractPage, but in summary it contains common functionality for all of our pages. For example, if our application has a navigational bar, global error messages, etc., this logic can be placed in a shared location.

The next thing you will notice is that we have a member variable for each of the parts of the HTML page that we are interested in. These are of type WebElement. WebDriver's PageFactory allows us to remove a lot of code from the HtmlUnit version of CreateMessagePage by automatically resolving each WebElement. The PageFactory#initElements(WebDriver,Class<T>) method will automatically resolve each WebElement by using the field name and looking it up by the id or name of the element within the HTML page.

We can use the @FindBy annotation to override the default lookup behavior. Our example demonstrates how to use the @FindBy annotation to look up our submit button using a css selector, input[type=submit].

Finally, we can verify that a new message was created successfully. The following assertions use the FEST assertion library.

assertThat(viewMessagePage.getMessage()).isEqualTo(expectedMessage);

assertThat(viewMessagePage.getSuccess()).isEqualTo("Successfully created a new message");

We can see that our ViewMessagePage allows us to interact with our custom domain model. For example, it exposes a method that returns a Message object.

public Message getMessage() throws ParseException {

Message message = new Message();

message.setId(getId());

message.setCreated(getCreated());

message.setSummary(getSummary());

message.setText(getText());

return message;

}

We can then leverage the rich domain objects in our assertions.

Lastly, don’t forget to close the WebDriver instance when the test is complete.

@After

public void destroy() {

if (driver != null) {

driver.close();

}

}

For additional information on using WebDriver, refer to the Selenium WebDriver documentation.

Advanced MockMvcHtmlUnitDriverBuilder

In the examples so far, we have used MockMvcHtmlUnitDriverBuilder in the simplest way possible, by building a WebDriver based on the WebApplicationContext loaded for us by the Spring TestContext Framework. This approach is repeated here.

@Autowired

WebApplicationContext context;

WebDriver driver;

@Before

public void setup() {

driver = MockMvcHtmlUnitDriverBuilder

.webAppContextSetup(context)

.build();

}

We can also specify additional configuration options.

WebDriver driver;

@Before

public void setup() {

driver = MockMvcHtmlUnitDriverBuilder

// demonstrates applying a MockMvcConfigurer (Spring Security)

.webAppContextSetup(context, springSecurity())

// for illustration only - defaults to ""

.contextPath("")

// By default MockMvc is used for localhost only;

// the following will use MockMvc for example.com and example.org as well

.useMockMvcForHosts("example.com","example.org")

.build();

}

As an alternative, we can perform the exact same setup by configuring the MockMvc instance separately and supplying it to the MockMvcHtmlUnitDriverBuilder as follows.

MockMvc mockMvc = MockMvcBuilders

.webAppContextSetup(context)

.apply(springSecurity())

.build();

driver = MockMvcHtmlUnitDriverBuilder

.mockMvcSetup(mockMvc)

// for illustration only - defaults to ""

.contextPath("")

// By default MockMvc is used for localhost only;

// the following will use MockMvc for example.com and example.org as well

.useMockMvcForHosts("example.com","example.org")

.build();

This is more verbose, but by building the WebDriver with a MockMvc instance we have the full power of MockMvc at our fingertips.

For additional information on creating a MockMvc instance refer to Setup Choices.

MockMvc and Geb

In the previous section, we saw how to use MockMvc with WebDriver. In this section, we will use Geb to make our tests even Groovy-er.

Why Geb and MockMvc?

Geb is backed by WebDriver, so it offers many of the same benefits that we get from WebDriver. However, Geb makes things even easier by taking care of some of the boilerplate code for us.

MockMvc and Geb Setup

We can easily initialize a Geb Browser with a Selenium WebDriver that uses MockMvc as follows.

def setup() {

browser.driver = MockMvcHtmlUnitDriverBuilder

.webAppContextSetup(context)

.build()

}

This is a simple example of using MockMvcHtmlUnitDriverBuilder. For more advanced usage, refer to Advanced MockMvcHtmlUnitDriverBuilder

This will ensure that any URL referencing localhost as the server will be directed to our MockMvc instance without the need for a real HTTP connection. Any other URL will be requested using a network connection as normal. This allows us to easily test the use of CDNs.

MockMvc and Geb Usage

Now we can use Geb as we normally would, but without the need to deploy our application to a Servlet container. For example, we can request the view to create a message with the following:

to CreateMessagePage

We can then fill out the form and submit it to create a message.

when:

form.summary = expectedSummary

form.text = expectedMessage

submit.click(ViewMessagePage)

Any unrecognized method calls or property accesses/references that are not found will be forwarded to the current page object. This removes a lot of the boilerplate code we needed when using WebDriver directly.

As with direct WebDriver usage, this improves on the design of our HtmlUnit test by leveraging the Page Object Pattern. As mentioned previously, we can use the Page Object Pattern with HtmlUnit and WebDriver, but it is even easier with Geb. Let’s take a look at our new Groovy-based CreateMessagePage implementation.

class CreateMessagePage extends Page {

static url = 'messages/form'

static at = { assert title == 'Messages : Create'; true }

static content = {

submit { $('input[type=submit]') }

form { $('form') }

errors(required:false) { $('label.error, .alert-error')?.text() }

}

}

The first thing you will notice is that our CreateMessagePage extends Page. We won’t go over the details of Page, but in summary it contains common functionality for all of our pages. The next thing you will notice is that we define a URL in which this page can be found. This allows us to navigate to the page as follows.

to CreateMessagePage

We also have an at closure that determines if we are at the specified page. It should return true if we are on the correct page. This is why we can assert that we are on the correct page as follows.

then:

at CreateMessagePage

errors.contains('This field is required.')

We use an assertion in the closure, so that we can determine where things went wrong if we were at the wrong page.

Next we create a content closure that specifies all the areas of interest within the page. We can use a jQuery-ish Navigator API to select the content we are interested in.

Finally, we can verify that a new message was created successfully.

then:

at ViewMessagePage

success == 'Successfully created a new message'

id

date

summary == expectedSummary

message == expectedMessage

For further details on how to get the most out of Geb, consult The Book of Geb user’s manual.

3.6.3. Client-Side REST Tests

Client-side tests can be used to test code that internally uses the RestTemplate. The idea is to declare expected requests and to provide "stub" responses so that you can focus on testing the code in isolation, i.e. without running a server. Here is an example:

RestTemplate restTemplate = new RestTemplate();

MockRestServiceServer mockServer = MockRestServiceServer.bindTo(restTemplate).build();

mockServer.expect(requestTo("/greeting")).andRespond(withSuccess());

// Test code that uses the above RestTemplate ...

mockServer.verify();

In the above example, MockRestServiceServer, the central class for client-side REST tests, configures the RestTemplate with a custom ClientHttpRequestFactory that asserts actual requests against expectations and returns "stub" responses. In this case we expect a request to "/greeting" and want to return a 200 response with "text/plain" content. We could define as additional expected requests and stub responses as needed. When expected requests and stub responses are defined, the RestTemplate can be used in client-side code as usual. At the end of testing mockServer.verify() can be used to verify that all expectations have been satisfied.

By default requests are expected in the order in which expectations were declared. You can set the ignoreExpectOrder option when building the server in which case all expectations are checked (in order) to find a match for a given request. That means requests are allowed to come in any order. Here is an example:

server = MockRestServiceServer.bindTo(restTemplate).ignoreExpectOrder(true).build();

Even with unordered requests by default each request is allowed to execute once only. The expect method provides an overloaded variant that accepts an ExpectedCount argument that specifies a count range, e.g. once, manyTimes, max, min, between, and so on. Here is an example:

RestTemplate restTemplate = new RestTemplate();

MockRestServiceServer mockServer = MockRestServiceServer.bindTo(restTemplate).build();

mockServer.expect(times(2), requestTo("/foo")).andRespond(withSuccess());

mockServer.expect(times(3), requestTo("/bar")).andRespond(withSuccess());

// ...

mockServer.verify();

Note that when ignoreExpectOrder is not set (the default), and therefore requests are expected in order of declaration, then that order only applies to the first of any expected request. For example if "/foo" is expected 2 times followed by "/bar" 3 times, then there should be a request to "/foo" before there is a request to "/bar" but aside from that subsequent "/foo" and "/bar" requests can come at any time.

As an alternative to all of the above the client-side test support also provides a ClientHttpRequestFactory implementation that can be configured into a RestTemplate to bind it to a MockMvc instance. That allows processing requests using actual server-side logic but without running a server. Here is an example:

MockMvc mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build();

this.restTemplate = new RestTemplate(new MockMvcClientHttpRequestFactory(mockMvc));

// Test code that uses the above RestTemplate ...

mockServer.verify();

Static Imports

Just like with server-side tests, the fluent API for client-side tests requires a few static imports. Those are easy to find by searching "MockRest\*". Eclipse users should add "MockRestRequestMatchers.\*" and "MockRestResponseCreators.\*" as "favorite static members" in the Eclipse preferences under Java → Editor → Content Assist → Favorites. That allows using content assist after typing the first character of the static method name. Other IDEs (e.g. IntelliJ) may not require any additional configuration. Just check the support for code completion on static members.

Further Examples of Client-side REST Tests

Spring MVC Test’s own tests include example tests of client-side REST tests.

3.7. WebTestClient

WebTestClient is a thin shell around WebClient, using it to perform requests and exposing a dedicated, fluent API for verifying responses. WebTestClient bind to a WebFlux application using a mock request and response, or it can test any web server over an HTTP connection.

Kotlin users, please see this section related to use of the WebTestClient.

3.7.1. Setup

To create a WebTestClient you must choose one of several server setup options. Effectively you’re either configuring the WebFlux application to bind to, or using a URL to connect to a running server.

Bind to controller

Use this server setup to test one @Controller at a time:

client = WebTestClient.bindToController(new TestController()).build();

The above loads the WebFlux Java config and registers the given controller. The resulting WebFlux application will be tested without an HTTP server using mock request and response objects. There are more methods on the builder to customize the default WebFlux Java config.

Bind to RouterFunction

Use this option to set up a server from a RouterFunction:

RouterFunction<?> route = ...

client = WebTestClient.bindToRouterFunction(route).build();

Internally the provided configuration is passed to RouterFunctions.toWebHandler. The resulting WebFlux application will be tested without an HTTP server using mock request and response objects.

Bind to ApplicationContext

Use this option to setup a server from the Spring configuration of your application, or some subset of it:

@RunWith(SpringRunner.class)

@ContextConfiguration(classes = WebConfig.class)

public class MyTests {

@Autowired

private ApplicationContext context;

private WebTestClient client;

@Before

public void setUp() {

client = WebTestClient.bindToApplicationContext(context).build();

}

}

Specify the configuration to load

Inject the configuration

Create the WebTestClient

Internally the provided configuration is passed to WebHttpHandlerBuilder to set up the request processing chain, see WebHandler API for more details. The resulting WebFlux application will be tested without an HTTP server using mock request and response objects.

Bind to server

This server setup option allows you to connect to a running server:

client = WebTestClient.bindToServer().baseUrl("http://localhost:8080").build();

Client builder

In addition to the server setup options above, you can also configure client options including base URL, default headers, client filters, and others. These options are readily available following bindToServer. For all others, you need to use configureClient() to transition from server to client configuration as shown below:

client = WebTestClient.bindToController(new TestController())

.configureClient()

.baseUrl("/test")

.build();

3.7.2. Writing tests

WebTestClient is a thin shell around WebClient. It provides an identical API up to the point of performing a request via exchange(). What follows after exchange() is a chained API workflow to verify responses.

Typically you start by asserting the response status and headers:

client.get().uri("/persons/1")

.accept(MediaType.APPLICATION\_JSON\_UTF8)

.exchange()

.expectStatus().isOk()

.expectHeader().contentType(MediaType.APPLICATION\_JSON\_UTF8)

// ...

Then you specify how to decode and consume the response body:

expectBody(Class<T>) — decode to single object.

expectBodyList(Class<T>) — decode and collect objects to List<T>.

expectBody() — decode to byte[] for JSON content or empty body.

Then you can use built-in assertions for the body. Here is one example:

client.get().uri("/persons")

.exchange()

.expectStatus().isOk()

.expectBodyList(Person.class).hasSize(3).contains(person);

You can go beyond the built-in assertions and create your own:

client.get().uri("/persons/1")

.exchange()

.expectStatus().isOk()

.expectBody(Person.class)

.consumeWith(result -> {

// custom assertions (e.g. AssertJ)...

});

You can also exit the workflow and get a result:

EntityExchangeResult<Person> result = client.get().uri("/persons/1")

.exchange()

.expectStatus().isOk()

.expectBody(Person.class)

.returnResult();

When you need to decode to a target type with generics, look for the overloaded methods that accept ParameterizedTypeReference instead of Class<T>.

No content

If the response has no content, or you don’t care if it does, use Void.class which ensures that resources are released:

client.get().uri("/persons/123")

.exchange()

.expectStatus().isNotFound()

.expectBody(Void.class);

Or if you want to assert there is no response content, use this:

client.post().uri("/persons")

.body(personMono, Person.class)

.exchange()

.expectStatus().isCreated()

.expectBody().isEmpty();

JSON content

When you use expectBody() the response is consumed as a byte[]. This is useful for raw content assertions. For example you can use JSONAssert to verify JSON content:

client.get().uri("/persons/1")

.exchange()

.expectStatus().isOk()

.expectBody()

.json("{\"name\":\"Jane\"}")

You can also use JSONPath expressions:

client.get().uri("/persons")

.exchange()

.expectStatus().isOk()

.expectBody()

.jsonPath("$[0].name").isEqualTo("Jane")

.jsonPath("$[1].name").isEqualTo("Jason");

Streaming responses

To test infinite streams (e.g. "text/event-stream", "application/stream+json"), you’ll need to exit the chained API, via returnResult, immediately after response status and header assertions, as shown below:

FluxExchangeResult<MyEvent> result = client.get().uri("/events")

.accept(TEXT\_EVENT\_STREAM)

.exchange()

.expectStatus().isOk()

.returnResult(MyEvent.class);

Now you can consume the Flux<T>, assert decoded objects as they come, and then cancel at some point when test objects are met. We recommend using the StepVerifier from the reactor-test module to do that, for example:

Flux<Event> eventFux = result.getResponseBody();

StepVerifier.create(eventFlux)

.expectNext(person)

.expectNextCount(4)

.consumeNextWith(p -> ...)

.thenCancel()

.verify();

Request body

When it comes to building requests, the WebTestClient offers an identical API as the WebClient and the implementation is mostly a simple pass-through. Please refer to the WebClient documentation for examples on how to prepare a request with a body including submitting form data, multipart requests, and more.

3.8. PetClinic Example

The PetClinic application, available on GitHub, illustrates several features of the Spring TestContext Framework in a JUnit 4 environment. Most test functionality is included in the AbstractClinicTests, for which a partial listing is shown below:

import static org.junit.Assert.assertEquals;

// import ...

@ContextConfiguration

public abstract class AbstractClinicTests extends AbstractTransactionalJUnit4SpringContextTests {

@Autowired

protected Clinic clinic;

@Test

public void getVets() {

Collection<Vet> vets = this.clinic.getVets();

assertEquals("JDBC query must show the same number of vets",

super.countRowsInTable("VETS"), vets.size());

Vet v1 = EntityUtils.getById(vets, Vet.class, 2);

assertEquals("Leary", v1.getLastName());

assertEquals(1, v1.getNrOfSpecialties());

assertEquals("radiology", (v1.getSpecialties().get(0)).getName());

// ...

}

// ...

}

Notes:

This test case extends the AbstractTransactionalJUnit4SpringContextTests class, from which it inherits configuration for Dependency Injection (through the DependencyInjectionTestExecutionListener) and transactional behavior (through the TransactionalTestExecutionListener).

The clinic instance variable — the application object being tested — is set by Dependency Injection through @Autowired semantics.

The getVets() method illustrates how you can use the inherited countRowsInTable() method to easily verify the number of rows in a given table, thus verifying correct behavior of the application code being tested. This allows for stronger tests and lessens dependency on the exact test data. For example, you can add additional rows in the database without breaking tests.

Like many integration tests that use a database, most of the tests in AbstractClinicTests depend on a minimum amount of data already in the database before the test cases run. Alternatively, you might choose to populate the database within the test fixture set up of your test cases — again, within the same transaction as the tests.

The PetClinic application supports three data access technologies: JDBC, Hibernate, and JPA. By declaring @ContextConfiguration without any specific resource locations, the AbstractClinicTests class will have its application context loaded from the default location, AbstractClinicTests-context.xml, which declares a common DataSource. Subclasses specify additional context locations that must declare a PlatformTransactionManager and a concrete implementation of Clinic.

For example, the Hibernate implementation of the PetClinic tests contains the following implementation. For this example, HibernateClinicTests does not contain a single line of code: we only need to declare @ContextConfiguration, and the tests are inherited from AbstractClinicTests. Because @ContextConfiguration is declared without any specific resource locations, the Spring TestContext Framework loads an application context from all the beans defined in AbstractClinicTests-context.xml (i.e., the inherited locations) and HibernateClinicTests-context.xml, with HibernateClinicTests-context.xml possibly overriding beans defined in AbstractClinicTests-context.xml.

@ContextConfiguration

public class HibernateClinicTests extends AbstractClinicTests { }

In a large-scale application, the Spring configuration is often split across multiple files. Consequently, configuration locations are typically specified in a common base class for all application-specific integration tests. Such a base class may also add useful instance variables — populated by Dependency Injection, naturally — such as a SessionFactory in the case of an application using Hibernate.

As far as possible, you should have exactly the same Spring configuration files in your integration tests as in the deployed environment. One likely point of difference concerns database connection pooling and transaction infrastructure. If you are deploying to a full-blown application server, you will probably use its connection pool (available through JNDI) and JTA implementation. Thus in production you will use a JndiObjectFactoryBean or <jee:jndi-lookup> for the DataSource and JtaTransactionManager. JNDI and JTA will not be available in out-of-container integration tests, so you should use a combination like the Commons DBCP BasicDataSource and DataSourceTransactionManager or HibernateTransactionManager for them. You can factor out this variant behavior into a single XML file, having the choice between application server and a 'local' configuration separated from all other configuration, which will not vary between the test and production environments. In addition, it is advisable to use properties files for connection settings. See the PetClinic application for an example.

4. Further Resources

Consult the following resources for more information about testing:

JUnit: "A programmer-oriented testing framework for Java". Used by the Spring Framework in its test suite.

TestNG: A testing framework inspired by JUnit with added support for annotations, test groups, data-driven testing, distributed testing, etc.

AssertJ: "Fluent assertions for Java" including support for Java 8 lambdas, streams, etc.

Mock Objects: Article in Wikipedia.

MockObjects.com: Web site dedicated to mock objects, a technique for improving the design of code within test-driven development.

Mockito: Java mock library based on the test spy pattern.

EasyMock: Java library "that provides Mock Objects for interfaces (and objects through the class extension) by generating them on the fly using Java’s proxy mechanism." Used by the Spring Framework in its test suite.

JMock: Library that supports test-driven development of Java code with mock objects.

DbUnit: JUnit extension (also usable with Ant and Maven) targeted for database-driven projects that, among other things, puts your database into a known state between test runs.

The Grinder: Java load testing framework.

## Data Access

Version 5.0.8.RELEASE

This part of the reference documentation is concerned with data access and the interaction between the data access layer and the business or service layer.

Spring’s comprehensive transaction management support is covered in some detail, followed by thorough coverage of the various data access frameworks and technologies that the Spring Framework integrates with.

1. Transaction Management

1.1. Introduction to Spring Framework transaction management

Comprehensive transaction support is among the most compelling reasons to use the Spring Framework. The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits:

Consistent programming model across different transaction APIs such as Java Transaction API (JTA), JDBC, Hibernate, and Java Persistence API (JPA).

Support for declarative transaction management.

Simpler API for programmatic transaction management than complex transaction APIs such as JTA.

Excellent integration with Spring’s data access abstractions.

The following sections describe the Spring Framework’s transaction value-adds and technologies. (The chapter also includes discussions of best practices, application server integration, and solutions to common problems.)

Advantages of the Spring Framework’s transaction support model describes why you would use the Spring Framework’s transaction abstraction instead of EJB Container-Managed Transactions (CMT) or choosing to drive local transactions through a proprietary API such as Hibernate.

Understanding the Spring Framework transaction abstraction outlines the core classes and describes how to configure and obtain DataSource instances from a variety of sources.

Synchronizing resources with transactions describes how the application code ensures that resources are created, reused, and cleaned up properly.

Declarative transaction management describes support for declarative transaction management.

Programmatic transaction management covers support for programmatic (that is, explicitly coded) transaction management.

Transaction bound event describes how you could use application events within a transaction.

1.2. Advantages of the Spring Framework’s transaction support model

Traditionally, Java EE developers have had two choices for transaction management: global or local transactions, both of which have profound limitations. Global and local transaction management is reviewed in the next two sections, followed by a discussion of how the Spring Framework’s transaction management support addresses the limitations of the global and local transaction models.

1.2.1. Global transactions

Global transactions enable you to work with multiple transactional resources, typically relational databases and message queues. The application server manages global transactions through the JTA, which is a cumbersome API to use (partly due to its exception model). Furthermore, a JTA UserTransaction normally needs to be sourced from JNDI, meaning that you also need to use JNDI in order to use JTA. Obviously the use of global transactions would limit any potential reuse of application code, as JTA is normally only available in an application server environment.

Previously, the preferred way to use global transactions was via EJB CMT (Container Managed Transaction): CMT is a form of declarative transaction management (as distinguished from programmatic transaction management). EJB CMT removes the need for transaction-related JNDI lookups, although of course the use of EJB itself necessitates the use of JNDI. It removes most but not all of the need to write Java code to control transactions. The significant downside is that CMT is tied to JTA and an application server environment. Also, it is only available if one chooses to implement business logic in EJBs, or at least behind a transactional EJB facade. The negatives of EJB in general are so great that this is not an attractive proposition, especially in the face of compelling alternatives for declarative transaction management.

1.2.2. Local transactions

Local transactions are resource-specific, such as a transaction associated with a JDBC connection. Local transactions may be easier to use, but have significant disadvantages: they cannot work across multiple transactional resources. For example, code that manages transactions using a JDBC connection cannot run within a global JTA transaction. Because the application server is not involved in transaction management, it cannot help ensure correctness across multiple resources. (It is worth noting that most applications use a single transaction resource.) Another downside is that local transactions are invasive to the programming model.

1.2.3. Spring Framework’s consistent programming model

Spring resolves the disadvantages of global and local transactions. It enables application developers to use a consistent programming model in any environment. You write your code once, and it can benefit from different transaction management strategies in different environments. The Spring Framework provides both declarative and programmatic transaction management. Most users prefer declarative transaction management, which is recommended in most cases.

With programmatic transaction management, developers work with the Spring Framework transaction abstraction, which can run over any underlying transaction infrastructure. With the preferred declarative model, developers typically write little or no code related to transaction management, and hence do not depend on the Spring Framework transaction API, or any other transaction API.

Do you need an application server for transaction management?

The Spring Framework’s transaction management support changes traditional rules as to when an enterprise Java application requires an application server.

In particular, you do not need an application server simply for declarative transactions through EJBs. In fact, even if your application server has powerful JTA capabilities, you may decide that the Spring Framework’s declarative transactions offer more power and a more productive programming model than EJB CMT.

Typically you need an application server’s JTA capability only if your application needs to handle transactions across multiple resources, which is not a requirement for many applications. Many high-end applications use a single, highly scalable database (such as Oracle RAC) instead. Standalone transaction managers such as Atomikos Transactions and JOTM are other options. Of course, you may need other application server capabilities such as Java Message Service (JMS) and Java EE Connector Architecture (JCA).

The Spring Framework gives you the choice of when to scale your application to a fully loaded application server. Gone are the days when the only alternative to using EJB CMT or JTA was to write code with local transactions such as those on JDBC connections, and face a hefty rework if you need that code to run within global, container-managed transactions. With the Spring Framework, only some of the bean definitions in your configuration file, rather than your code, need to change.

1.3. Understanding the Spring Framework transaction abstraction

The key to the Spring transaction abstraction is the notion of a transaction strategy. A transaction strategy is defined by the org.springframework.transaction.PlatformTransactionManager interface:

public interface PlatformTransactionManager {

TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;

void commit(TransactionStatus status) throws TransactionException;

void rollback(TransactionStatus status) throws TransactionException;

}

This is primarily a service provider interface (SPI), although it can be used programmatically from your application code. Because PlatformTransactionManager is an interface, it can be easily mocked or stubbed as necessary. It is not tied to a lookup strategy such as JNDI. PlatformTransactionManager implementations are defined like any other object (or bean) in the Spring Framework IoC container. This benefit alone makes Spring Framework transactions a worthwhile abstraction even when you work with JTA. Transactional code can be tested much more easily than if it used JTA directly.

Again in keeping with Spring’s philosophy, the TransactionException that can be thrown by any of the PlatformTransactionManager interface’s methods is unchecked (that is, it extends the java.lang.RuntimeException class). Transaction infrastructure failures are almost invariably fatal. In rare cases where application code can actually recover from a transaction failure, the application developer can still choose to catch and handle TransactionException. The salient point is that developers are not forced to do so.

The getTransaction(..) method returns a TransactionStatus object, depending on a TransactionDefinition parameter. The returned TransactionStatus might represent a new transaction, or can represent an existing transaction if a matching transaction exists in the current call stack. The implication in this latter case is that, as with Java EE transaction contexts, a TransactionStatus is associated with a thread of execution.

The TransactionDefinition interface specifies:

Propagation: Typically, all code executed within a transaction scope will run in that transaction. However, you have the option of specifying the behavior in the event that a transactional method is executed when a transaction context already exists. For example, code can continue running in the existing transaction (the common case); or the existing transaction can be suspended and a new transaction created. Spring offers all of the transaction propagation options familiar from EJB CMT. To read about the semantics of transaction propagation in Spring, see Transaction propagation.

Isolation: The degree to which this transaction is isolated from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?

Timeout: How long this transaction runs before timing out and being rolled back automatically by the underlying transaction infrastructure.

Read-only status: A read-only transaction can be used when your code reads but does not modify data. Read-only transactions can be a useful optimization in some cases, such as when you are using Hibernate.

These settings reflect standard transactional concepts. If necessary, refer to resources that discuss transaction isolation levels and other core transaction concepts. Understanding these concepts is essential to using the Spring Framework or any transaction management solution.

The TransactionStatus interface provides a simple way for transactional code to control transaction execution and query transaction status. The concepts should be familiar, as they are common to all transaction APIs:

public interface TransactionStatus extends SavepointManager {

boolean isNewTransaction();

boolean hasSavepoint();

void setRollbackOnly();

boolean isRollbackOnly();

void flush();

boolean isCompleted();

}

Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct PlatformTransactionManager implementation is absolutely essential. You typically define this implementation through dependency injection.

PlatformTransactionManager implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate, and so on. The following examples show how you can define a local PlatformTransactionManager implementation. (This example works with plain JDBC.)

You define a JDBC DataSource

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="${jdbc.driverClassName}" />

<property name="url" value="${jdbc.url}" />

<property name="username" value="${jdbc.username}" />

<property name="password" value="${jdbc.password}" />

</bean>

The related PlatformTransactionManager bean definition will then have a reference to the DataSource definition. It will look like this:

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

If you use JTA in a Java EE container then you use a container DataSource, obtained through JNDI, in conjunction with Spring’s JtaTransactionManager. This is what the JTA and JNDI lookup version would look like:

<?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:jee="http://www.springframework.org/schema/jee"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/jee

http://www.springframework.org/schema/jee/spring-jee.xsd">

<jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>

<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager" />

<!-- other <bean/> definitions here -->

</beans>

The JtaTransactionManager does not need to know about the DataSource, or any other specific resources, because it uses the container’s global transaction management infrastructure.

The above definition of the dataSource bean uses the <jndi-lookup/> tag from the jee namespace. For more information see The JEE schema.

You can also use Hibernate local transactions easily, as shown in the following examples. In this case, you need to define a Hibernate LocalSessionFactoryBean, which your application code will use to obtain Hibernate Session instances.

The DataSource bean definition will be similar to the local JDBC example shown previously and thus is not shown in the following example.

If the DataSource, used by any non-JTA transaction manager, is looked up via JNDI and managed by a Java EE container, then it should be non-transactional because the Spring Framework, rather than the Java EE container, will manage the transactions.

The txManager bean in this case is of the HibernateTransactionManager type. In the same way as the DataSourceTransactionManager needs a reference to the DataSource, the HibernateTransactionManager needs a reference to the SessionFactory.

<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

<property name="dataSource" ref="dataSource"/>

<property name="mappingResources">

<list>

<value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>

</list>

</property>

<property name="hibernateProperties">

<value>

hibernate.dialect=${hibernate.dialect}

</value>

</property>

</bean>

<bean id="txManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">

<property name="sessionFactory" ref="sessionFactory"/>

</bean>

If you are using Hibernate and Java EE container-managed JTA transactions, then you should simply use the same JtaTransactionManager as in the previous JTA example for JDBC.

<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>

If you use JTA , then your transaction manager definition will look the same regardless of what data access technology you use, be it JDBC, Hibernate JPA or any other supported technology. This is due to the fact that JTA transactions are global transactions, which can enlist any transactional resource.

In all these cases, application code does not need to change. You can change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa.

1.4. Synchronizing resources with transactions

It should now be clear how you create different transaction managers, and how they are linked to related resources that need to be synchronized to transactions (for example DataSourceTransactionManager to a JDBC DataSource, HibernateTransactionManager to a Hibernate SessionFactory, and so forth). This section describes how the application code, directly or indirectly using a persistence API such as JDBC, Hibernate, or JPA, ensures that these resources are created, reused, and cleaned up properly. The section also discusses how transaction synchronization is triggered (optionally) through the relevant PlatformTransactionManager.

1.4.1. High-level synchronization approach

The preferred approach is to use Spring’s highest level template based persistence integration APIs or to use native ORM APIs with transaction- aware factory beans or proxies for managing the native resource factories. These transaction-aware solutions internally handle resource creation and reuse, cleanup, optional transaction synchronization of the resources, and exception mapping. Thus user data access code does not have to address these tasks, but can be focused purely on non-boilerplate persistence logic. Generally, you use the native ORM API or take a template approach for JDBC access by using the JdbcTemplate. These solutions are detailed in subsequent chapters of this reference documentation.

1.4.2. Low-level synchronization approach

Classes such as DataSourceUtils (for JDBC), EntityManagerFactoryUtils (for JPA), SessionFactoryUtils (for Hibernate), and so on exist at a lower level. When you want the application code to deal directly with the resource types of the native persistence APIs, you use these classes to ensure that proper Spring Framework-managed instances are obtained, transactions are (optionally) synchronized, and exceptions that occur in the process are properly mapped to a consistent API.

For example, in the case of JDBC, instead of the traditional JDBC approach of calling the getConnection() method on the DataSource, you instead use Spring’s org.springframework.jdbc.datasource.DataSourceUtils class as follows:

Connection conn = DataSourceUtils.getConnection(dataSource);

If an existing transaction already has a connection synchronized (linked) to it, that instance is returned. Otherwise, the method call triggers the creation of a new connection, which is (optionally) synchronized to any existing transaction, and made available for subsequent reuse in that same transaction. As mentioned, any SQLException is wrapped in a Spring Framework CannotGetJdbcConnectionException, one of the Spring Framework’s hierarchy of unchecked DataAccessExceptions. This approach gives you more information than can be obtained easily from the SQLException, and ensures portability across databases, even across different persistence technologies.

This approach also works without Spring transaction management (transaction synchronization is optional), so you can use it whether or not you are using Spring for transaction management.

Of course, once you have used Spring’s JDBC support, JPA support or Hibernate support, you will generally prefer not to use DataSourceUtils or the other helper classes, because you will be much happier working through the Spring abstraction than directly with the relevant APIs. For example, if you use the Spring JdbcTemplate or jdbc.object package to simplify your use of JDBC, correct connection retrieval occurs behind the scenes and you won’t need to write any special code.

1.4.3. TransactionAwareDataSourceProxy

At the very lowest level exists the TransactionAwareDataSourceProxy class. This is a proxy for a target DataSource, which wraps the target DataSource to add awareness of Spring-managed transactions. In this respect, it is similar to a transactional JNDI DataSource as provided by a Java EE server.

It should almost never be necessary or desirable to use this class, except when existing code must be called and passed a standard JDBC DataSource interface implementation. In that case, it is possible that this code is usable, but participating in Spring managed transactions. It is preferable to write your new code by using the higher level abstractions mentioned above.

1.5. Declarative transaction management

Most Spring Framework users choose declarative transaction management. This option has the least impact on application code, and hence is most consistent with the ideals of a non-invasive lightweight container.

The Spring Framework’s declarative transaction management is made possible with Spring aspect-oriented programming (AOP), although, as the transactional aspects code comes with the Spring Framework distribution and may be used in a boilerplate fashion, AOP concepts do not generally have to be understood to make effective use of this code.

The Spring Framework’s declarative transaction management is similar to EJB CMT in that you can specify transaction behavior (or lack of it) down to individual method level. It is possible to make a setRollbackOnly() call within a transaction context if necessary. The differences between the two types of transaction management are:

Unlike EJB CMT, which is tied to JTA, the Spring Framework’s declarative transaction management works in any environment. It can work with JTA transactions or local transactions using JDBC, JPA or Hibernate by simply adjusting the configuration files.

You can apply the Spring Framework declarative transaction management to any class, not merely special classes such as EJBs.

The Spring Framework offers declarative rollback rules, a feature with no EJB equivalent. Both programmatic and declarative support for rollback rules is provided.

The Spring Framework enables you to customize transactional behavior, by using AOP. For example, you can insert custom behavior in the case of transaction rollback. You can also add arbitrary advice, along with the transactional advice. With EJB CMT, you cannot influence the container’s transaction management except with setRollbackOnly().

The Spring Framework does not support propagation of transaction contexts across remote calls, as do high-end application servers. If you need this feature, we recommend that you use EJB. However, consider carefully before using such a feature, because normally, one does not want transactions to span remote calls.

Where is TransactionProxyFactoryBean?

Declarative transaction configuration in versions of Spring 2.0 and above differs considerably from previous versions of Spring. The main difference is that there is no longer any need to configure TransactionProxyFactoryBean beans.

The pre-Spring 2.0 configuration style is still 100% valid configuration; think of the new <tx:tags/> as simply defining TransactionProxyFactoryBean beans on your behalf.

The concept of rollback rules is important: they enable you to specify which exceptions (and throwables) should cause automatic rollback. You specify this declaratively, in configuration, not in Java code. So, although you can still call setRollbackOnly() on the TransactionStatus object to roll back the current transaction back, most often you can specify a rule that MyApplicationException must always result in rollback. The significant advantage to this option is that business objects do not depend on the transaction infrastructure. For example, they typically do not need to import Spring transaction APIs or other Spring APIs.

Although EJB container default behavior automatically rolls back the transaction on a system exception (usually a runtime exception), EJB CMT does not roll back the transaction automatically on anapplication exception (that is, a checked exception other than java.rmi.RemoteException). While the Spring default behavior for declarative transaction management follows EJB convention (roll back is automatic only on unchecked exceptions), it is often useful to customize this behavior.

1.5.1. Understanding the Spring Framework’s declarative transaction implementation

It is not sufficient to tell you simply to annotate your classes with the @Transactional annotation, add @EnableTransactionManagement to your configuration, and then expect you to understand how it all works. This section explains the inner workings of the Spring Framework’s declarative transaction infrastructure in the event of transaction-related issues.

The most important concepts to grasp with regard to the Spring Framework’s declarative transaction support are that this support is enabled via AOP proxies, and that the transactional advice is driven by metadata (currently XML- or annotation-based). The combination of AOP with transactional metadata yields an AOP proxy that uses a TransactionInterceptor in conjunction with an appropriate PlatformTransactionManager implementation to drive transactions around method invocations.

Spring AOP is covered in the AOP section.

Conceptually, calling a method on a transactional proxy looks like this…​

tx

1.5.2. Example of declarative transaction implementation

Consider the following interface, and its attendant implementation. This example uses Foo and Bar classes as placeholders so that you can concentrate on the transaction usage without focusing on a particular domain model. For the purposes of this example, the fact that the DefaultFooService class throws UnsupportedOperationException instances in the body of each implemented method is good; it allows you to see transactions created and then rolled back in response to the UnsupportedOperationException instance.

// the service interface that we want to make transactional

package x.y.service;

public interface FooService {

Foo getFoo(String fooName);

Foo getFoo(String fooName, String barName);

void insertFoo(Foo foo);

void updateFoo(Foo foo);

}

// an implementation of the above interface

package x.y.service;

public class DefaultFooService implements FooService {

public Foo getFoo(String fooName) {

throw new UnsupportedOperationException();

}

public Foo getFoo(String fooName, String barName) {

throw new UnsupportedOperationException();

}

public void insertFoo(Foo foo) {

throw new UnsupportedOperationException();

}

public void updateFoo(Foo foo) {

throw new UnsupportedOperationException();

}

}

Assume that the first two methods of the FooService interface, getFoo(String) and getFoo(String, String), must execute in the context of a transaction with read-only semantics, and that the other methods, insertFoo(Foo) and updateFoo(Foo), must execute in the context of a transaction with read-write semantics. The following configuration is explained in detail in the next few paragraphs.

<!-- from the file 'context.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:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- this is the service object that we want to make transactional -->

<bean id="fooService" class="x.y.service.DefaultFooService"/>

<!-- the transactional advice (what 'happens'; see the <aop:advisor/> bean below) -->

<tx:advice id="txAdvice" transaction-manager="txManager">

<!-- the transactional semantics... -->

<tx:attributes>

<!-- all methods starting with 'get' are read-only -->

<tx:method name="get\*" read-only="true"/>

<!-- other methods use the default transaction settings (see below) -->

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

<!-- ensure that the above transactional advice runs for any execution

of an operation defined by the FooService interface -->

<aop:config>

<aop:pointcut id="fooServiceOperation" expression="execution(\* x.y.service.FooService.\*(..))"/>

<aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>

</aop:config>

<!-- don't forget the DataSource -->

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>

<property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>

<property name="username" value="scott"/>

<property name="password" value="tiger"/>

</bean>

<!-- similarly, don't forget the PlatformTransactionManager -->

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

<!-- other <bean/> definitions here -->

</beans>

Examine the preceding configuration. You want to make a service object, the fooService bean, transactional. The transaction semantics to apply are encapsulated in the <tx:advice/> definition. The <tx:advice/> definition reads as "…​ all methods on starting with 'get' are to execute in the context of a read-only transaction, and all other methods are to execute with the default transaction semantics". The transaction-manager attribute of the <tx:advice/> tag is set to the name of the PlatformTransactionManager bean that is going to drive the transactions, in this case, the txManager bean.

You can omit the transaction-manager attribute in the transactional advice (<tx:advice/>) if the bean name of the PlatformTransactionManager that you want to wire in has the name transactionManager. If the PlatformTransactionManager bean that you want to wire in has any other name, then you must use the transaction-manager attribute explicitly, as in the preceding example.

The <aop:config/> definition ensures that the transactional advice defined by the txAdvice bean executes at the appropriate points in the program. First you define a pointcut that matches the execution of any operation defined in the FooService interface ( fooServiceOperation). Then you associate the pointcut with the txAdvice using an advisor. The result indicates that at the execution of a fooServiceOperation, the advice defined by txAdvice will be run.

The expression defined within the <aop:pointcut/> element is an AspectJ pointcut expression; see the AOP section for more details on pointcut expressions in Spring.

A common requirement is to make an entire service layer transactional. The best way to do this is simply to change the pointcut expression to match any operation in your service layer. For example:

<aop:config>

<aop:pointcut id="fooServiceMethods" expression="execution(\* x.y.service.\*.\*(..))"/>

<aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceMethods"/>

</aop:config>

In this example it is assumed that all your service interfaces are defined in the x.y.service package; see the AOP section for more details.

Now that we’ve analyzed the configuration, you may be asking yourself, "Okay…​ but what does all this configuration actually do?".

The above configuration will be used to create a transactional proxy around the object that is created from the fooService bean definition. The proxy will be configured with the transactional advice, so that when an appropriate method is invoked on the proxy, a transaction is started, suspended, marked as read-only, and so on, depending on the transaction configuration associated with that method. Consider the following program that test drives the above configuration:

public final class Boot {

public static void main(final String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml", Boot.class);

FooService fooService = (FooService) ctx.getBean("fooService");

fooService.insertFoo (new Foo());

}

}

The output from running the preceding program will resemble the following. (The Log4J output and the stack trace from the UnsupportedOperationException thrown by the insertFoo(..) method of the DefaultFooService class have been truncated for clarity.)

<!-- the Spring container is starting up... -->

[AspectJInvocationContextExposingAdvisorAutoProxyCreator] - Creating implicit proxy for bean 'fooService' with 0 common interceptors and 1 specific interceptors

<!-- the DefaultFooService is actually proxied -->

[JdkDynamicAopProxy] - Creating JDK dynamic proxy for [x.y.service.DefaultFooService]

<!-- ... the insertFoo(..) method is now being invoked on the proxy -->

[TransactionInterceptor] - Getting transaction for x.y.service.FooService.insertFoo

<!-- the transactional advice kicks in here... -->

[DataSourceTransactionManager] - Creating new transaction with name [x.y.service.FooService.insertFoo]

[DataSourceTransactionManager] - Acquired Connection [org.apache.commons.dbcp.PoolableConnection@a53de4] for JDBC transaction

<!-- the insertFoo(..) method from DefaultFooService throws an exception... -->

[RuleBasedTransactionAttribute] - Applying rules to determine whether transaction should rollback on java.lang.UnsupportedOperationException

[TransactionInterceptor] - Invoking rollback for transaction on x.y.service.FooService.insertFoo due to throwable [java.lang.UnsupportedOperationException]

<!-- and the transaction is rolled back (by default, RuntimeException instances cause rollback) -->

[DataSourceTransactionManager] - Rolling back JDBC transaction on Connection [org.apache.commons.dbcp.PoolableConnection@a53de4]

[DataSourceTransactionManager] - Releasing JDBC Connection after transaction

[DataSourceUtils] - Returning JDBC Connection to DataSource

Exception in thread "main" java.lang.UnsupportedOperationException at x.y.service.DefaultFooService.insertFoo(DefaultFooService.java:14)

<!-- AOP infrastructure stack trace elements removed for clarity -->

at $Proxy0.insertFoo(Unknown Source)

at Boot.main(Boot.java:11)

1.5.3. Rolling back a declarative transaction

The previous section outlined the basics of how to specify transactional settings for classes, typically service layer classes, declaratively in your application. This section describes how you can control the rollback of transactions in a simple declarative fashion.

The recommended way to indicate to the Spring Framework’s transaction infrastructure that a transaction’s work is to be rolled back is to throw an Exception from code that is currently executing in the context of a transaction. The Spring Framework’s transaction infrastructure code will catch any unhandled Exception as it bubbles up the call stack, and make a determination whether to mark the transaction for rollback.

In its default configuration, the Spring Framework’s transaction infrastructure code only marks a transaction for rollback in the case of runtime, unchecked exceptions; that is, when the thrown exception is an instance or subclass of RuntimeException. ( Errors will also - by default - result in a rollback). Checked exceptions that are thrown from a transactional method do not result in rollback in the default configuration.

You can configure exactly which Exception types mark a transaction for rollback, including checked exceptions. The following XML snippet demonstrates how you configure rollback for a checked, application-specific Exception type.

<tx:advice id="txAdvice" transaction-manager="txManager">

<tx:attributes>

<tx:method name="get\*" read-only="true" rollback-for="NoProductInStockException"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

You can also specify 'no rollback rules', if you do not want a transaction rolled back when an exception is thrown. The following example tells the Spring Framework’s transaction infrastructure to commit the attendant transaction even in the face of an unhandled InstrumentNotFoundException.

<tx:advice id="txAdvice">

<tx:attributes>

<tx:method name="updateStock" no-rollback-for="InstrumentNotFoundException"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

When the Spring Framework’s transaction infrastructure catches an exception and it consults configured rollback rules to determine whether to mark the transaction for rollback, the strongest matching rule wins. So in the case of the following configuration, any exception other than an InstrumentNotFoundException results in a rollback of the attendant transaction.

<tx:advice id="txAdvice">

<tx:attributes>

<tx:method name="\*" rollback-for="Throwable" no-rollback-for="InstrumentNotFoundException"/>

</tx:attributes>

</tx:advice>

You can also indicate a required rollback programmatically. Although very simple, this process is quite invasive, and tightly couples your code to the Spring Framework’s transaction infrastructure:

public void resolvePosition() {

try {

// some business logic...

} catch (NoProductInStockException ex) {

// trigger rollback programmatically

TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();

}

}

You are strongly encouraged to use the declarative approach to rollback if at all possible. Programmatic rollback is available should you absolutely need it, but its usage flies in the face of achieving a clean POJO-based architecture.

1.5.4. Configuring different transactional semantics for different beans

Consider the scenario where you have a number of service layer objects, and you want to apply a totally different transactional configuration to each of them. You do this by defining distinct <aop:advisor/> elements with differing pointcut and advice-ref attribute values.

As a point of comparison, first assume that all of your service layer classes are defined in a root x.y.service package. To make all beans that are instances of classes defined in that package (or in subpackages) and that have names ending in Service have the default transactional configuration, you would write the following:

<?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:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<aop:config>

<aop:pointcut id="serviceOperation"

expression="execution(\* x.y.service..\*Service.\*(..))"/>

<aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice"/>

</aop:config>

<!-- these two beans will be transactional... -->

<bean id="fooService" class="x.y.service.DefaultFooService"/>

<bean id="barService" class="x.y.service.extras.SimpleBarService"/>

<!-- ... and these two beans won't -->

<bean id="anotherService" class="org.xyz.SomeService"/> <!-- (not in the right package) -->

<bean id="barManager" class="x.y.service.SimpleBarManager"/> <!-- (doesn't end in 'Service') -->

<tx:advice id="txAdvice">

<tx:attributes>

<tx:method name="get\*" read-only="true"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

<!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->

</beans>

The following example shows how to configure two distinct beans with totally different transactional settings.

<?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:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<aop:config>

<aop:pointcut id="defaultServiceOperation"

expression="execution(\* x.y.service.\*Service.\*(..))"/>

<aop:pointcut id="noTxServiceOperation"

expression="execution(\* x.y.service.ddl.DefaultDdlManager.\*(..))"/>

<aop:advisor pointcut-ref="defaultServiceOperation" advice-ref="defaultTxAdvice"/>

<aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice"/>

</aop:config>

<!-- this bean will be transactional (see the 'defaultServiceOperation' pointcut) -->

<bean id="fooService" class="x.y.service.DefaultFooService"/>

<!-- this bean will also be transactional, but with totally different transactional settings -->

<bean id="anotherFooService" class="x.y.service.ddl.DefaultDdlManager"/>

<tx:advice id="defaultTxAdvice">

<tx:attributes>

<tx:method name="get\*" read-only="true"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

<tx:advice id="noTxAdvice">

<tx:attributes>

<tx:method name="\*" propagation="NEVER"/>

</tx:attributes>

</tx:advice>

<!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->

</beans>

1.5.5. <tx:advice/> settings

This section summarizes the various transactional settings that can be specified using the <tx:advice/> tag. The default <tx:advice/> settings are:

Propagation setting is REQUIRED.

Isolation level is DEFAULT.

Transaction is read/write.

Transaction timeout defaults to the default timeout of the underlying transaction system, or none if timeouts are not supported.

Any RuntimeException triggers rollback, and any checked Exception does not.

You can change these default settings; the various attributes of the <tx:method/> tags that are nested within <tx:advice/> and <tx:attributes/> tags are summarized below:

Table 1. <tx:method/> settings

Attribute Required? Default Description

name

Yes

Method name(s) with which the transaction attributes are to be associated. The wildcard (\*) character can be used to associate the same transaction attribute settings with a number of methods; for example, get\*, handle\*, on\*Event, and so forth.

propagation

No

REQUIRED

Transaction propagation behavior.

isolation

No

DEFAULT

Transaction isolation level. Only applicable to propagation REQUIRED or REQUIRES\_NEW.

timeout

No

-1

Transaction timeout (seconds). Only applicable to propagation REQUIRED or REQUIRES\_NEW.

read-only

No

false

Read/write vs. read-only transaction. Only applicable to REQUIRED or REQUIRES\_NEW.

rollback-for

No

Exception(s) that trigger rollback; comma-delimited. For example, com.foo.MyBusinessException,ServletException.

no-rollback-for

No

Exception(s) that do not trigger rollback; comma-delimited. For example, com.foo.MyBusinessException,ServletException.

1.5.6. Using @Transactional

In addition to the XML-based declarative approach to transaction configuration, you can use an annotation-based approach. Declaring transaction semantics directly in the Java source code puts the declarations much closer to the affected code. There is not much danger of undue coupling, because code that is meant to be used transactionally is almost always deployed that way anyway.

The standard javax.transaction.Transactional annotation is also supported as a drop-in replacement to Spring’s own annotation. Please refer to JTA 1.2 documentation for more details.

The ease-of-use afforded by the use of the @Transactional annotation is best illustrated with an example, which is explained in the text that follows. Consider the following class definition:

// the service class that we want to make transactional

@Transactional

public class DefaultFooService implements FooService {

Foo getFoo(String fooName);

Foo getFoo(String fooName, String barName);

void insertFoo(Foo foo);

void updateFoo(Foo foo);

}

When the above POJO is defined as a bean in a Spring IoC container, the bean instance can be made transactional by adding merely one line of XML configuration:

<!-- from the file 'context.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:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- this is the service object that we want to make transactional -->

<bean id="fooService" class="x.y.service.DefaultFooService"/>

<!-- enable the configuration of transactional behavior based on annotations -->

<tx:annotation-driven transaction-manager="txManager"/><!-- a PlatformTransactionManager is still required -->

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<!-- (this dependency is defined somewhere else) -->

<property name="dataSource" ref="dataSource"/>

</bean>

<!-- other <bean/> definitions here -->

</beans>

You can omit the transaction-manager attribute in the <tx:annotation-driven/> tag if the bean name of the PlatformTransactionManager that you want to wire in has the name transactionManager. If the PlatformTransactionManager bean that you want to dependency-inject has any other name, then you have to use the transaction-manager attribute explicitly, as in the preceding example.

The @EnableTransactionManagement annotation provides equivalent support if you are using Java based configuration. Simply add the annotation to a @Configuration class. See the javadocs for full details.

Method visibility and @Transactional

When using proxies, you should apply the @Transactional annotation only to methods with public visibility. If you do annotate protected, private or package-visible methods with the @Transactional annotation, no error is raised, but the annotated method does not exhibit the configured transactional settings. Consider the use of AspectJ (see below) if you need to annotate non-public methods.

You can place the @Transactional annotation before an interface definition, a method on an interface, a class definition, or a public method on a class. However, the mere presence of the @Transactional annotation is not enough to activate the transactional behavior. The @Transactional annotation is simply metadata that can be consumed by some runtime infrastructure that is @Transactional-aware and that can use the metadata to configure the appropriate beans with transactional behavior. In the preceding example, the <tx:annotation-driven/> element switches on the transactional behavior.

Spring recommends that you only annotate concrete classes (and methods of concrete classes) with the @Transactional annotation, as opposed to annotating interfaces. You certainly can place the @Transactional annotation on an interface (or an interface method), but this works only as you would expect it to if you are using interface-based proxies. The fact that Java annotations are not inherited from interfaces means that if you are using class-based proxies ( proxy-target-class="true") or the weaving-based aspect ( mode="aspectj"), then the transaction settings are not recognized by the proxying and weaving infrastructure, and the object will not be wrapped in a transactional proxy, which would be decidedly bad.

In proxy mode (which is the default), only external method calls coming in through the proxy are intercepted. This means that self-invocation, in effect, a method within the target object calling another method of the target object, will not lead to an actual transaction at runtime even if the invoked method is marked with @Transactional. Also, the proxy must be fully initialized to provide the expected behaviour so you should not rely on this feature in your initialization code, i.e. @PostConstruct.

Consider the use of AspectJ mode (see mode attribute in table below) if you expect self-invocations to be wrapped with transactions as well. In this case, there will not be a proxy in the first place; instead, the target class will be weaved (that is, its byte code will be modified) in order to turn @Transactional into runtime behavior on any kind of method.

Table 2. Annotation driven transaction settings

XML Attribute Annotation Attribute Default Description

transaction-manager

N/A (See TransactionManagementConfigurer javadocs)

transactionManager

Name of transaction manager to use. Only required if the name of the transaction manager is not transactionManager, as in the example above.

mode

mode

proxy

The default mode "proxy" processes annotated beans to be proxied using Spring’s AOP framework (following proxy semantics, as discussed above, applying to method calls coming in through the proxy only). The alternative mode "aspectj" instead weaves the affected classes with Spring’s AspectJ transaction aspect, modifying the target class byte code to apply to any kind of method call. AspectJ weaving requires spring-aspects.jar in the classpath as well as load-time weaving (or compile-time weaving) enabled. (See Spring configuration for details on how to set up load-time weaving.)

proxy-target-class

proxyTargetClass

false

Applies to proxy mode only. Controls what type of transactional proxies are created for classes annotated with the @Transactional annotation. If the proxy-target-class attribute is set to true, then class-based proxies are created. If proxy-target-class is false or if the attribute is omitted, then standard JDK interface-based proxies are created. (See Proxying mechanisms for a detailed examination of the different proxy types.)

order

order

Ordered.LOWEST\_PRECEDENCE

Defines the order of the transaction advice that is applied to beans annotated with @Transactional. (For more information about the rules related to ordering of AOP advice, see Advice ordering.) No specified ordering means that the AOP subsystem determines the order of the advice.

The default advice mode for processing @Transactional annotations is "proxy" which allows for interception of calls through the proxy only; local calls within the same class cannot get intercepted that way. For a more advanced mode of interception, consider switching to "aspectj" mode in combination with compile/load-time weaving.

The proxy-target-class attribute controls what type of transactional proxies are created for classes annotated with the @Transactional annotation. If proxy-target-class is set to true, class-based proxies are created. If proxy-target-class is false or if the attribute is omitted, standard JDK interface-based proxies are created. (See [aop-proxying] for a discussion of the different proxy types.)

@EnableTransactionManagement and <tx:annotation-driven/> only looks for @Transactional on beans in the same application context they are defined in. This means that, if you put annotation driven configuration in a WebApplicationContext for a DispatcherServlet, it only checks for @Transactional beans in your controllers, and not your services. See MVC for more information.

The most derived location takes precedence when evaluating the transactional settings for a method. In the case of the following example, the DefaultFooService class is annotated at the class level with the settings for a read-only transaction, but the @Transactional annotation on the updateFoo(Foo) method in the same class takes precedence over the transactional settings defined at the class level.

@Transactional(readOnly = true)

public class DefaultFooService implements FooService {

public Foo getFoo(String fooName) {

// do something

}

// these settings have precedence for this method

@Transactional(readOnly = false, propagation = Propagation.REQUIRES\_NEW)

public void updateFoo(Foo foo) {

// do something

}

}

@Transactional settings

The @Transactional annotation is metadata that specifies that an interface, class, or method must have transactional semantics; for example, "start a brand new read-only transaction when this method is invoked, suspending any existing transaction". The default @Transactional settings are as follows:

Propagation setting is PROPAGATION\_REQUIRED.

Isolation level is ISOLATION\_DEFAULT.

Transaction is read/write.

Transaction timeout defaults to the default timeout of the underlying transaction system, or to none if timeouts are not supported.

Any RuntimeException triggers rollback, and any checked Exception does not.

These default settings can be changed; the various properties of the @Transactional annotation are summarized in the following table:

Table 3. @Transactional Settings

Property Type Description

value

String

Optional qualifier specifying the transaction manager to be used.

propagation

enum: Propagation

Optional propagation setting.

isolation

enum: Isolation

Optional isolation level. Only applicable to propagation REQUIRED or REQUIRES\_NEW.

timeout

int (in seconds granularity)

Optional transaction timeout. Only applicable to propagation REQUIRED or REQUIRES\_NEW.

readOnly

boolean

Read/write vs. read-only transaction. Only applicable to REQUIRED or REQUIRES\_NEW.

rollbackFor

Array of Class objects, which must be derived from Throwable.

Optional array of exception classes that must cause rollback.

rollbackForClassName

Array of class names. Classes must be derived from Throwable.

Optional array of names of exception classes that must cause rollback.

noRollbackFor

Array of Class objects, which must be derived from Throwable.

Optional array of exception classes that must not cause rollback.

noRollbackForClassName

Array of String class names, which must be derived from Throwable.

Optional array of names of exception classes that must not cause rollback.

Currently you cannot have explicit control over the name of a transaction, where 'name' means the transaction name that will be shown in a transaction monitor, if applicable (for example, WebLogic’s transaction monitor), and in logging output. For declarative transactions, the transaction name is always the fully-qualified class name + "." + method name of the transactionally-advised class. For example, if the handlePayment(..) method of the BusinessService class started a transaction, the name of the transaction would be: com.foo.BusinessService.handlePayment.

Multiple Transaction Managers with @Transactional

Most Spring applications only need a single transaction manager, but there may be situations where you want multiple independent transaction managers in a single application. The value attribute of the @Transactional annotation can be used to optionally specify the identity of the PlatformTransactionManager to be used. This can either be the bean name or the qualifier value of the transaction manager bean. For example, using the qualifier notation, the following Java code

public class TransactionalService {

@Transactional("order")

public void setSomething(String name) { ... }

@Transactional("account")

public void doSomething() { ... }

}

could be combined with the following transaction manager bean declarations in the application context.

<tx:annotation-driven/>

<bean id="transactionManager1" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

...

<qualifier value="order"/>

</bean>

<bean id="transactionManager2" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

...

<qualifier value="account"/>

</bean>

In this case, the two methods on TransactionalService will run under separate transaction managers, differentiated by the "order" and "account" qualifiers. The default <tx:annotation-driven> target bean name transactionManager will still be used if no specifically qualified PlatformTransactionManager bean is found.

Custom shortcut annotations

If you find you are repeatedly using the same attributes with @Transactional on many different methods, then Spring’s meta-annotation support allows you to define custom shortcut annotations for your specific use cases. For example, defining the following annotations

@Target({ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Transactional("order")

public @interface OrderTx {

}

@Target({ElementType.METHOD, ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Transactional("account")

public @interface AccountTx {

}

allows us to write the example from the previous section as

public class TransactionalService {

@OrderTx

public void setSomething(String name) { ... }

@AccountTx

public void doSomething() { ... }

}

Here we have used the syntax to define the transaction manager qualifier, but could also have included propagation behavior, rollback rules, timeouts etc.

1.5.7. Transaction propagation

This section describes some semantics of transaction propagation in Spring. Please note that this section is not an introduction to transaction propagation proper; rather it details some of the semantics regarding transaction propagation in Spring.

In Spring-managed transactions, be aware of the difference between physical and logical transactions, and how the propagation setting applies to this difference.

Required

tx prop required

PROPAGATION\_REQUIRED

PROPAGATION\_REQUIRED enforces a physical transaction: either locally for the current scope if no transaction exists yet, or participating in an existing 'outer' transaction defined for a larger scope. This is a fine default in common call stack arrangements within the same thread, e.g. a service facade delegating to several repository methods where all the underlying resources have to participate in the service-level transaction.

By default, a participating transaction will join the characteristics of the outer scope, silently ignoring the local isolation level, timeout value or read-only flag (if any). Consider switching the "validateExistingTransactions" flag to "true" on your transaction manager if you’d like isolation level declarations to get rejected when participating in an existing transaction with a different isolation level. This non-lenient mode will also reject read-only mismatches, i.e. an inner read-write transaction trying to participate in a read-only outer scope.

When the propagation setting is PROPAGATION\_REQUIRED, a logical transaction scope is created for each method upon which the setting is applied. Each such logical transaction scope can determine rollback-only status individually, with an outer transaction scope being logically independent from the inner transaction scope. Of course, in case of standard PROPAGATION\_REQUIRED behavior, all these scopes will be mapped to the same physical transaction. So a rollback-only marker set in the inner transaction scope does affect the outer transaction’s chance to actually commit (as you would expect it to).

However, in the case where an inner transaction scope sets the rollback-only marker, the outer transaction has not decided on the rollback itself, and so the rollback (silently triggered by the inner transaction scope) is unexpected. A corresponding UnexpectedRollbackException is thrown at that point. This is expected behavior so that the caller of a transaction can never be misled to assume that a commit was performed when it really was not. So if an inner transaction (of which the outer caller is not aware) silently marks a transaction as rollback-only, the outer caller still calls commit. The outer caller needs to receive an UnexpectedRollbackException to indicate clearly that a rollback was performed instead.

RequiresNew

tx prop requires new

PROPAGATION\_REQUIRES\_NEW

PROPAGATION\_REQUIRES\_NEW, in contrast to PROPAGATION\_REQUIRED, always uses an independent physical transaction for each affected transaction scope, never participating in an existing transaction for an outer scope. In such an arrangement, the underlying resource transactions are different and hence can commit or roll back independently, with an outer transaction not affected by an inner transaction’s rollback status, and with an inner transaction’s locks released immediately after its completion. Such an independent inner transaction may also declare its own isolation level, timeout and read-only settings, never inheriting an outer transaction’s characteristics.

Nested

PROPAGATION\_NESTED uses a single physical transaction with multiple savepoints that it can roll back to. Such partial rollbacks allow an inner transaction scope to trigger a rollback for its scope, with the outer transaction being able to continue the physical transaction despite some operations having been rolled back. This setting is typically mapped onto JDBC savepoints, so will only work with JDBC resource transactions. See Spring’s DataSourceTransactionManager.

1.5.8. Advising transactional operations

Suppose you want to execute both transactional and some basic profiling advice. How do you effect this in the context of <tx:annotation-driven/>?

When you invoke the updateFoo(Foo) method, you want to see the following actions:

Configured profiling aspect starts up.

Transactional advice executes.

Method on the advised object executes.

Transaction commits.

Profiling aspect reports exact duration of the whole transactional method invocation.

This chapter is not concerned with explaining AOP in any great detail (except as it applies to transactions). See AOP for detailed coverage of the following AOP configuration and AOP in general.

Here is the code for a simple profiling aspect discussed above. The ordering of advice is controlled through the Ordered interface. For full details on advice ordering, see Advice ordering. .

package x.y;

import org.aspectj.lang.ProceedingJoinPoint;

import org.springframework.util.StopWatch;

import org.springframework.core.Ordered;

public class SimpleProfiler implements Ordered {

private int order;

// allows us to control the ordering of advice

public int getOrder() {

return this.order;

}

public void setOrder(int order) {

this.order = order;

}

// this method is the around advice

public Object profile(ProceedingJoinPoint call) throws Throwable {

Object returnValue;

StopWatch clock = new StopWatch(getClass().getName());

try {

clock.start(call.toShortString());

returnValue = call.proceed();

} finally {

clock.stop();

System.out.println(clock.prettyPrint());

}

return returnValue;

}

}

<?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:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<bean id="fooService" class="x.y.service.DefaultFooService"/>

<!-- this is the aspect -->

<bean id="profiler" class="x.y.SimpleProfiler">

<!-- execute before the transactional advice (hence the lower order number) -->

<property name="order" value="1"/>

</bean>

<tx:annotation-driven transaction-manager="txManager" order="200"/>

<aop:config>

<!-- this advice will execute around the transactional advice -->

<aop:aspect id="profilingAspect" ref="profiler">

<aop:pointcut id="serviceMethodWithReturnValue"

expression="execution(!void x.y..\*Service.\*(..))"/>

<aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>

</aop:aspect>

</aop:config>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>

<property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>

<property name="username" value="scott"/>

<property name="password" value="tiger"/>

</bean>

<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource"/>

</bean>

</beans>

The result of the above configuration is a fooService bean that has profiling and transactional aspects applied to it in the desired order. You configure any number of additional aspects in similar fashion.

The following example effects the same setup as above, but uses the purely XML declarative approach.

<?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:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<bean id="fooService" class="x.y.service.DefaultFooService"/>

<!-- the profiling advice -->

<bean id="profiler" class="x.y.SimpleProfiler">

<!-- execute before the transactional advice (hence the lower order number) -->

<property name="order" value="1"/>

</bean>

<aop:config>

<aop:pointcut id="entryPointMethod" expression="execution(\* x.y..\*Service.\*(..))"/>

<!-- will execute after the profiling advice (c.f. the order attribute) -->

<aop:advisor advice-ref="txAdvice" pointcut-ref="entryPointMethod" order="2"/>

<!-- order value is higher than the profiling aspect -->

<aop:aspect id="profilingAspect" ref="profiler">

<aop:pointcut id="serviceMethodWithReturnValue"

expression="execution(!void x.y..\*Service.\*(..))"/>

<aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>

</aop:aspect>

</aop:config>

<tx:advice id="txAdvice" transaction-manager="txManager">

<tx:attributes>

<tx:method name="get\*" read-only="true"/>

<tx:method name="\*"/>

</tx:attributes>

</tx:advice>

<!-- other <bean/> definitions such as a DataSource and a PlatformTransactionManager here -->

</beans>

The result of the above configuration will be a fooService bean that has profiling and transactional aspects applied to it in that order. If you want the profiling advice to execute after the transactional advice on the way in, and before the transactional advice on the way out, then you simply swap the value of the profiling aspect bean’s order property so that it is higher than the transactional advice’s order value.

You configure additional aspects in similar fashion.

1.5.9. Using @Transactional with AspectJ

It is also possible to use the Spring Framework’s @Transactional support outside of a Spring container by means of an AspectJ aspect. To do so, you first annotate your classes (and optionally your classes' methods) with the @Transactional annotation, and then you link (weave) your application with the org.springframework.transaction.aspectj.AnnotationTransactionAspect defined in the spring-aspects.jar file. The aspect must also be configured with a transaction manager. You can of course use the Spring Framework’s IoC container to take care of dependency-injecting the aspect. The simplest way to configure the transaction management aspect is to use the <tx:annotation-driven/> element and specify the mode attribute to aspectj as described in Using @Transactional. Because we’re focusing here on applications running outside of a Spring container, we’ll show you how to do it programmatically.

Prior to continuing, you may want to read Using @Transactional and AOP respectively.

// construct an appropriate transaction manager

DataSourceTransactionManager txManager = new DataSourceTransactionManager(getDataSource());

// configure the AnnotationTransactionAspect to use it; this must be done before executing any transactional methods

AnnotationTransactionAspect.aspectOf().setTransactionManager(txManager);

When using this aspect, you must annotate the implementation class (and/or methods within that class), not the interface (if any) that the class implements. AspectJ follows Java’s rule that annotations on interfaces are not inherited.

The @Transactional annotation on a class specifies the default transaction semantics for the execution of any public method in the class.

The @Transactional annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). Any method may be annotated, regardless of visibility.

To weave your applications with the AnnotationTransactionAspect you must either build your application with AspectJ (see the AspectJ Development Guide) or use load-time weaving. See Load-time weaving with AspectJ in the Spring Framework for a discussion of load-time weaving with AspectJ.

1.6. Programmatic transaction management

The Spring Framework provides two means of programmatic transaction management:

Using the TransactionTemplate.

Using a PlatformTransactionManager implementation directly.

The Spring team generally recommends the TransactionTemplate for programmatic transaction management. The second approach is similar to using the JTA UserTransaction API, although exception handling is less cumbersome.

1.6.1. Using the TransactionTemplate

The TransactionTemplate adopts the same approach as other Spring templates such as the JdbcTemplate. It uses a callback approach, to free application code from having to do the boilerplate acquisition and release of transactional resources, and results in code that is intention driven, in that the code that is written focuses solely on what the developer wants to do.

As you will see in the examples that follow, using the TransactionTemplate absolutely couples you to Spring’s transaction infrastructure and APIs. Whether or not programmatic transaction management is suitable for your development needs is a decision that you will have to make yourself.

Application code that must execute in a transactional context, and that will use the TransactionTemplate explicitly, looks like the following. You, as an application developer, write a TransactionCallback implementation (typically expressed as an anonymous inner class) that contains the code that you need to execute in the context of a transaction. You then pass an instance of your custom TransactionCallback to the execute(..) method exposed on the TransactionTemplate.

public class SimpleService implements Service {

// single TransactionTemplate shared amongst all methods in this instance

private final TransactionTemplate transactionTemplate;

// use constructor-injection to supply the PlatformTransactionManager

public SimpleService(PlatformTransactionManager transactionManager) {

this.transactionTemplate = new TransactionTemplate(transactionManager);

}

public Object someServiceMethod() {

return transactionTemplate.execute(new TransactionCallback() {

// the code in this method executes in a transactional context

public Object doInTransaction(TransactionStatus status) {

updateOperation1();

return resultOfUpdateOperation2();

}

});

}

}

If there is no return value, use the convenient TransactionCallbackWithoutResult class with an anonymous class as follows:

transactionTemplate.execute(new TransactionCallbackWithoutResult() {

protected void doInTransactionWithoutResult(TransactionStatus status) {

updateOperation1();

updateOperation2();

}

});

Code within the callback can roll the transaction back by calling the setRollbackOnly() method on the supplied TransactionStatus object:

transactionTemplate.execute(new TransactionCallbackWithoutResult() {

protected void doInTransactionWithoutResult(TransactionStatus status) {

try {

updateOperation1();

updateOperation2();

} catch (SomeBusinessException ex) {

status.setRollbackOnly();

}

}

});

Specifying transaction settings

You can specify transaction settings such as the propagation mode, the isolation level, the timeout, and so forth on the TransactionTemplate either programmatically or in configuration. TransactionTemplate instances by default have the default transactional settings. The following example shows the programmatic customization of the transactional settings for a specific TransactionTemplate:

public class SimpleService implements Service {

private final TransactionTemplate transactionTemplate;

public SimpleService(PlatformTransactionManager transactionManager) {

this.transactionTemplate = new TransactionTemplate(transactionManager);

// the transaction settings can be set here explicitly if so desired

this.transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION\_READ\_UNCOMMITTED);

this.transactionTemplate.setTimeout(30); // 30 seconds

// and so forth...

}

}

The following example defines a TransactionTemplate with some custom transactional settings, using Spring XML configuration. The sharedTransactionTemplate can then be injected into as many services as are required.

<bean id="sharedTransactionTemplate"

class="org.springframework.transaction.support.TransactionTemplate">

<property name="isolationLevelName" value="ISOLATION\_READ\_UNCOMMITTED"/>

<property name="timeout" value="30"/>

</bean>"

Finally, instances of the TransactionTemplate class are threadsafe, in that instances do not maintain any conversational state. TransactionTemplate instances do however maintain configuration state, so while a number of classes may share a single instance of a TransactionTemplate, if a class needs to use a TransactionTemplate with different settings (for example, a different isolation level), then you need to create two distinct TransactionTemplate instances.

1.6.2. Using the PlatformTransactionManager

You can also use the org.springframework.transaction.PlatformTransactionManager directly to manage your transaction. Simply pass the implementation of the PlatformTransactionManager you are using to your bean through a bean reference. Then, using the TransactionDefinition and TransactionStatus objects you can initiate transactions, roll back, and commit.

DefaultTransactionDefinition def = new DefaultTransactionDefinition();

// explicitly setting the transaction name is something that can only be done programmatically

def.setName("SomeTxName");

def.setPropagationBehavior(TransactionDefinition.PROPAGATION\_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);

try {

// execute your business logic here

}

catch (MyException ex) {

txManager.rollback(status);

throw ex;

}

txManager.commit(status);

1.7. Choosing between programmatic and declarative transaction management

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. For example, if you have a web application that require transactions only for certain update operations, you may not want to set up transactional proxies using Spring or any other technology. In this case, using the TransactionTemplate may be a good approach. Being able to set the transaction name explicitly is also something that can only be done using the programmatic approach to transaction management.

On the other hand, if your application has numerous transactional operations, declarative transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure. When using the Spring Framework, rather than EJB CMT, the configuration cost of declarative transaction management is greatly reduced.

1.8. Transaction bound event

As of Spring 4.2, the listener of an event can be bound to a phase of the transaction. The typical example is to handle the event when the transaction has completed successfully: this allows events to be used with more flexibility when the outcome of the current transaction actually matters to the listener.

Registering a regular event listener is done via the @EventListener annotation. If you need to bind it to the transaction use @TransactionalEventListener. When you do so, the listener will be bound to the commit phase of the transaction by default.

Let’s take an example to illustrate this concept. Assume that a component publishes an order created event and we want to define a listener that should only handle that event once the transaction in which it has been published has committed successfully:

@Component

public class MyComponent {

@TransactionalEventListener

public void handleOrderCreatedEvent(CreationEvent<Order> creationEvent) {

...

}

}

The TransactionalEventListener annotation exposes a phase attribute that allows us to customize which phase of the transaction the listener should be bound to. The valid phases are BEFORE\_COMMIT, AFTER\_COMMIT (default), AFTER\_ROLLBACK and AFTER\_COMPLETION that aggregates the transaction completion (be it a commit or a rollback).

If no transaction is running, the listener is not invoked at all since we can’t honor the required semantics. It is however possible to override that behaviour by setting the fallbackExecution attribute of the annotation to true.

1.9. Application server-specific integration

Spring’s transaction abstraction generally is application server agnostic. Additionally, Spring’s JtaTransactionManager class, which can optionally perform a JNDI lookup for the JTA UserTransaction and TransactionManager objects, autodetects the location for the latter object, which varies by application server. Having access to the JTA TransactionManager allows for enhanced transaction semantics, in particular supporting transaction suspension. See the JtaTransactionManager javadocs for details.

Spring’s JtaTransactionManager is the standard choice to run on Java EE application servers, and is known to work on all common servers. Advanced functionality such as transaction suspension works on many servers as well — including GlassFish, JBoss and Geronimo — without any special configuration required. However, for fully supported transaction suspension and further advanced integration, Spring ships special adapters for WebLogic Server and WebSphere. These adapters are discussed in the following sections.

For standard scenarios, including WebLogic Server and WebSphere, consider using the convenient <tx:jta-transaction-manager/> configuration element. When configured, this element automatically detects the underlying server and chooses the best transaction manager available for the platform. This means that you won’t have to configure server-specific adapter classes (as discussed in the following sections) explicitly; rather, they are chosen automatically, with the standard JtaTransactionManager as default fallback.

1.9.1. IBM WebSphere

On WebSphere 6.1.0.9 and above, the recommended Spring JTA transaction manager to use is WebSphereUowTransactionManager. This special adapter leverages IBM’s UOWManager API, which is available in WebSphere Application Server 6.1.0.9 and later. With this adapter, Spring-driven transaction suspension (suspend/resume as initiated by PROPAGATION\_REQUIRES\_NEW) is officially supported by IBM.

1.9.2. Oracle WebLogic Server

On WebLogic Server 9.0 or above, you typically would use the WebLogicJtaTransactionManager instead of the stock JtaTransactionManager class. This special WebLogic-specific subclass of the normal JtaTransactionManager supports the full power of Spring’s transaction definitions in a WebLogic-managed transaction environment, beyond standard JTA semantics: Features include transaction names, per-transaction isolation levels, and proper resuming of transactions in all cases.

1.10. Solutions to common problems

1.10.1. Use of the wrong transaction manager for a specific DataSource

Use the correct PlatformTransactionManager implementation based on your choice of transactional technologies and requirements. Used properly, the Spring Framework merely provides a straightforward and portable abstraction. If you are using global transactions, you must use the org.springframework.transaction.jta.JtaTransactionManager class (or an application server-specific subclass of it) for all your transactional operations. Otherwise the transaction infrastructure attempts to perform local transactions on resources such as container DataSource instances. Such local transactions do not make sense, and a good application server treats them as errors.

1.11. Further resources

For more information about the Spring Framework’s transaction support:

Distributed transactions in Spring, with and without XA is a JavaWorld presentation in which Spring’s David Syer guides you through seven patterns for distributed transactions in Spring applications, three of them with XA and four without.

Java Transaction Design Strategies is a book available from InfoQ that provides a well-paced introduction to transactions in Java. It also includes side-by-side examples of how to configure and use transactions with both the Spring Framework and EJB3.

2. DAO support

2.1. Introduction

The Data Access Object (DAO) support in Spring is aimed at making it easy to work with data access technologies like JDBC, Hibernate or JPA in a consistent way. This allows one to switch between the aforementioned persistence technologies fairly easily and it also allows one to code without worrying about catching exceptions that are specific to each technology.

2.2. Consistent exception hierarchy

Spring provides a convenient translation from technology-specific exceptions like SQLException to its own exception class hierarchy with the DataAccessException as the root exception. These exceptions wrap the original exception so there is never any risk that one might lose any information as to what might have gone wrong.

In addition to JDBC exceptions, Spring can also wrap Hibernate-specific exceptions, converting them to a set of focused runtime exceptions (the same is true for JPA exceptions). This allows one to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without having annoying boilerplate catch-and-throw blocks and exception declarations in one’s DAOs. (One can still trap and handle exceptions anywhere one needs to though.) As mentioned above, JDBC exceptions (including database-specific dialects) are also converted to the same hierarchy, meaning that one can perform some operations with JDBC within a consistent programming model.

The above holds true for the various template classes in Springs support for various ORM frameworks. If one uses the interceptor-based classes then the application must care about handling HibernateExceptions and PersistenceExceptions itself, preferably via delegating to SessionFactoryUtils’ `convertHibernateAccessException(..) or convertJpaAccessException() methods respectively. These methods convert the exceptions to ones that are compatible with the exceptions in the org.springframework.dao exception hierarchy. As PersistenceExceptions are unchecked, they can simply get thrown too, sacrificing generic DAO abstraction in terms of exceptions though.

The exception hierarchy that Spring provides can be seen below. (Please note that the class hierarchy detailed in the image shows only a subset of the entire DataAccessException hierarchy.)

DataAccessException

2.3. Annotations used for configuring DAO or Repository classes

The best way to guarantee that your Data Access Objects (DAOs) or repositories provide exception translation is to use the @Repository annotation. This annotation also allows the component scanning support to find and configure your DAOs and repositories without having to provide XML configuration entries for them.

@Repository

public class SomeMovieFinder implements MovieFinder {

// ...

}

Any DAO or repository implementation will need to access to a persistence resource, depending on the persistence technology used; for example, a JDBC-based repository will need access to a JDBC DataSource; a JPA-based repository will need access to an EntityManager. The easiest way to accomplish this is to have this resource dependency injected using one of the @Autowired,, @Inject, @Resource or @PersistenceContext annotations. Here is an example for a JPA repository:

@Repository

public class JpaMovieFinder implements MovieFinder {

@PersistenceContext

private EntityManager entityManager;

// ...

}

If you are using the classic Hibernate APIs than you can inject the SessionFactory:

@Repository

public class HibernateMovieFinder implements MovieFinder {

private SessionFactory sessionFactory;

@Autowired

public void setSessionFactory(SessionFactory sessionFactory) {

this.sessionFactory = sessionFactory;

}

// ...

}

Last example we will show here is for typical JDBC support. You would have the DataSource injected into an initialization method where you would create a JdbcTemplate and other data access support classes like SimpleJdbcCall etc using this DataSource.

@Repository

public class JdbcMovieFinder implements MovieFinder {

private JdbcTemplate jdbcTemplate;

@Autowired

public void init(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

// ...

}

Please see the specific coverage of each persistence technology for details on how to configure the application context to take advantage of these annotations.

3. Data access with JDBC

3.1. Introduction to Spring Framework’s JDBC support

The value-add provided by the Spring Framework JDBC abstraction is perhaps best shown by the sequence of actions outlined in the table below. The table shows what actions Spring will take care of and which actions are the responsibility of you, the application developer.

Table 4. Spring JDBC - who does what?

Action Spring You

Define connection parameters.

X

Open the connection.

X

Specify the SQL statement.

X

Declare parameters and provide parameter values

X

Prepare and execute the statement.

X

Set up the loop to iterate through the results (if any).

X

Do the work for each iteration.

X

Process any exception.

X

Handle transactions.

X

Close the connection, statement and resultset.

X

The Spring Framework takes care of all the low-level details that can make JDBC such a tedious API to develop with.

3.1.1. Choosing an approach for JDBC database access

You can choose among several approaches to form the basis for your JDBC database access. In addition to three flavors of the JdbcTemplate, a new SimpleJdbcInsert and SimplejdbcCall approach optimizes database metadata, and the RDBMS Object style takes a more object-oriented approach similar to that of JDO Query design. Once you start using one of these approaches, you can still mix and match to include a feature from a different approach. All approaches require a JDBC 2.0-compliant driver, and some advanced features require a JDBC 3.0 driver.

JdbcTemplate is the classic Spring JDBC approach and the most popular. This "lowest level" approach and all others use a JdbcTemplate under the covers.

NamedParameterJdbcTemplate wraps a JdbcTemplate to provide named parameters instead of the traditional JDBC "?" placeholders. This approach provides better documentation and ease of use when you have multiple parameters for an SQL statement.

SimpleJdbcInsert and SimpleJdbcCall optimize database metadata to limit the amount of necessary configuration. This approach simplifies coding so that you only need to provide the name of the table or procedure and provide a map of parameters matching the column names. This only works if the database provides adequate metadata. If the database doesn’t provide this metadata, you will have to provide explicit configuration of the parameters.

RDBMS Objects including MappingSqlQuery, SqlUpdate and StoredProcedure requires you to create reusable and thread-safe objects during initialization of your data access layer. This approach is modeled after JDO Query wherein you define your query string, declare parameters, and compile the query. Once you do that, execute methods can be called multiple times with various parameter values passed in.

3.1.2. Package hierarchy

The Spring Framework’s JDBC abstraction framework consists of four different packages, namely core, datasource, object, and support.

The org.springframework.jdbc.core package contains the JdbcTemplate class and its various callback interfaces, plus a variety of related classes. A subpackage named org.springframework.jdbc.core.simple contains the SimpleJdbcInsert and SimpleJdbcCall classes. Another subpackage named org.springframework.jdbc.core.namedparam contains the NamedParameterJdbcTemplate class and the related support classes. See Using the JDBC core classes to control basic JDBC processing and error handling, JDBC batch operations, and Simplifying JDBC operations with the SimpleJdbc classes.

The org.springframework.jdbc.datasource package contains a utility class for easy DataSource access, and various simple DataSource implementations that can be used for testing and running unmodified JDBC code outside of a Java EE container. A subpackage named org.springfamework.jdbc.datasource.embedded provides support for creating embedded databases using Java database engines such as HSQL, H2, and Derby. See Controlling database connections and Embedded database support.

The org.springframework.jdbc.object package contains classes that represent RDBMS queries, updates, and stored procedures as thread-safe, reusable objects. See Modeling JDBC operations as Java objects. This approach is modeled by JDO, although objects returned by queries are naturally disconnected from the database. This higher level of JDBC abstraction depends on the lower-level abstraction in the org.springframework.jdbc.core package.

The org.springframework.jdbc.support package provides SQLException translation functionality and some utility classes. Exceptions thrown during JDBC processing are translated to exceptions defined in the org.springframework.dao package. This means that code using the Spring JDBC abstraction layer does not need to implement JDBC or RDBMS-specific error handling. All translated exceptions are unchecked, which gives you the option of catching the exceptions from which you can recover while allowing other exceptions to be propagated to the caller. See SQLExceptionTranslator.

3.2. Using the JDBC core classes to control basic JDBC processing and error handling

3.2.1. JdbcTemplate

The JdbcTemplate class is the central class in the JDBC core package. It handles the creation and release of resources, which helps you avoid common errors such as forgetting to close the connection. It performs the basic tasks of the core JDBC workflow such as statement creation and execution, leaving application code to provide SQL and extract results. The JdbcTemplate class executes SQL queries, update statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

When you use the JdbcTemplate for your code, you only need to implement callback interfaces, giving them a clearly defined contract. The PreparedStatementCreator callback interface creates a prepared statement given a Connection provided by this class, providing SQL and any necessary parameters. The same is true for the CallableStatementCreator interface, which creates callable statements. The RowCallbackHandler interface extracts values from each row of a ResultSet.

The JdbcTemplate can be used within a DAO implementation through direct instantiation with a DataSource reference, or be configured in a Spring IoC container and given to DAOs as a bean reference.

The DataSource should always be configured as a bean in the Spring IoC container. In the first case the bean is given to the service directly; in the second case it is given to the prepared template.

All SQL issued by this class is logged at the DEBUG level under the category corresponding to the fully qualified class name of the template instance (typically JdbcTemplate, but it may be different if you are using a custom subclass of the JdbcTemplate class).

Examples of JdbcTemplate class usage

This section provides some examples of JdbcTemplate class usage. These examples are not an exhaustive list of all of the functionality exposed by the JdbcTemplate; see the attendant javadocs for that.

Querying (SELECT)

Here is a simple query for getting the number of rows in a relation:

int rowCount = this.jdbcTemplate.queryForObject("select count(\*) from t\_actor", Integer.class);

A simple query using a bind variable:

int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(

"select count(\*) from t\_actor where first\_name = ?", Integer.class, "Joe");

Querying for a String:

String lastName = this.jdbcTemplate.queryForObject(

"select last\_name from t\_actor where id = ?",

new Object[]{1212L}, String.class);

Querying and populating a single domain object:

Actor actor = this.jdbcTemplate.queryForObject(

"select first\_name, last\_name from t\_actor where id = ?",

new Object[]{1212L},

new RowMapper<Actor>() {

public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {

Actor actor = new Actor();

actor.setFirstName(rs.getString("first\_name"));

actor.setLastName(rs.getString("last\_name"));

return actor;

}

});

Querying and populating a number of domain objects:

List<Actor> actors = this.jdbcTemplate.query(

"select first\_name, last\_name from t\_actor",

new RowMapper<Actor>() {

public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {

Actor actor = new Actor();

actor.setFirstName(rs.getString("first\_name"));

actor.setLastName(rs.getString("last\_name"));

return actor;

}

});

If the last two snippets of code actually existed in the same application, it would make sense to remove the duplication present in the two RowMapper anonymous inner classes, and extract them out into a single class (typically a static nested class) that can then be referenced by DAO methods as needed. For example, it may be better to write the last code snippet as follows:

public List<Actor> findAllActors() {

return this.jdbcTemplate.query( "select first\_name, last\_name from t\_actor", new ActorMapper());

}

private static final class ActorMapper implements RowMapper<Actor> {

public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {

Actor actor = new Actor();

actor.setFirstName(rs.getString("first\_name"));

actor.setLastName(rs.getString("last\_name"));

return actor;

}

}

Updating (INSERT/UPDATE/DELETE) with JdbcTemplate

You use the update(..) method to perform insert, update and delete operations. Parameter values are usually provided as var args or alternatively as an object array.

this.jdbcTemplate.update(

"insert into t\_actor (first\_name, last\_name) values (?, ?)",

"Leonor", "Watling");

this.jdbcTemplate.update(

"update t\_actor set last\_name = ? where id = ?",

"Banjo", 5276L);

this.jdbcTemplate.update(

"delete from actor where id = ?",

Long.valueOf(actorId));

Other JdbcTemplate operations

You can use the execute(..) method to execute any arbitrary SQL, and as such the method is often used for DDL statements. It is heavily overloaded with variants taking callback interfaces, binding variable arrays, and so on.

this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");

The following example invokes a simple stored procedure. More sophisticated stored procedure support is covered later.

this.jdbcTemplate.update(

"call SUPPORT.REFRESH\_ACTORS\_SUMMARY(?)",

Long.valueOf(unionId));

JdbcTemplate best practices

Instances of the JdbcTemplate class are threadsafe once configured. This is important because it means that you can configure a single instance of a JdbcTemplate and then safely inject this shared reference into multiple DAOs (or repositories). The JdbcTemplate is stateful, in that it maintains a reference to a DataSource, but this state is not conversational state.

A common practice when using the JdbcTemplate class (and the associated NamedParameterJdbcTemplate classes) is to configure a DataSource in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes; the JdbcTemplate is created in the setter for the DataSource. This leads to DAOs that look in part like the following:

public class JdbcCorporateEventDao implements CorporateEventDao {

private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

// JDBC-backed implementations of the methods on the CorporateEventDao follow...

}

The corresponding configuration might look like this.

<?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">

<bean id="corporateEventDao" class="com.example.JdbcCorporateEventDao">

<property name="dataSource" ref="dataSource"/>

</bean>

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="${jdbc.driverClassName}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

<context:property-placeholder location="jdbc.properties"/>

</beans>

An alternative to explicit configuration is to use component-scanning and annotation support for dependency injection. In this case you annotate the class with @Repository (which makes it a candidate for component-scanning) and annotate the DataSource setter method with @Autowired.

@Repository

public class JdbcCorporateEventDao implements CorporateEventDao {

private JdbcTemplate jdbcTemplate;

@Autowired

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

// JDBC-backed implementations of the methods on the CorporateEventDao follow...

}

The corresponding XML configuration file would look like the following:

<?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">

<!-- Scans within the base package of the application for @Component classes to configure as beans -->

<context:component-scan base-package="org.springframework.docs.test" />

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="${jdbc.driverClassName}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

<context:property-placeholder location="jdbc.properties"/>

</beans>

If you are using Spring’s JdbcDaoSupport class, and your various JDBC-backed DAO classes extend from it, then your sub-class inherits a setDataSource(..) method from the JdbcDaoSupport class. You can choose whether to inherit from this class. The JdbcDaoSupport class is provided as a convenience only.

Regardless of which of the above template initialization styles you choose to use (or not), it is seldom necessary to create a new instance of a JdbcTemplate class each time you want to execute SQL. Once configured, a JdbcTemplate instance is threadsafe. You may want multiple JdbcTemplate instances if your application accesses multiple databases, which requires multiple DataSources, and subsequently multiple differently configured JdbcTemplates.

3.2.2. NamedParameterJdbcTemplate

The NamedParameterJdbcTemplate class adds support for programming JDBC statements using named parameters, as opposed to programming JDBC statements using only classic placeholder ( '?') arguments. The NamedParameterJdbcTemplate class wraps a JdbcTemplate, and delegates to the wrapped JdbcTemplate to do much of its work. This section describes only those areas of the NamedParameterJdbcTemplate class that differ from the JdbcTemplate itself; namely, programming JDBC statements using named parameters.

// some JDBC-backed DAO class...

private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);

}

public int countOfActorsByFirstName(String firstName) {

String sql = "select count(\*) from T\_ACTOR where first\_name = :first\_name";

SqlParameterSource namedParameters = new MapSqlParameterSource("first\_name", firstName);

return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);

}

Notice the use of the named parameter notation in the value assigned to the sql variable, and the corresponding value that is plugged into the namedParameters variable (of type MapSqlParameterSource).

Alternatively, you can pass along named parameters and their corresponding values to a NamedParameterJdbcTemplate instance by using the Map-based style.The remaining methods exposed by the NamedParameterJdbcOperations and implemented by the NamedParameterJdbcTemplate class follow a similar pattern and are not covered here.

The following example shows the use of the Map-based style.

// some JDBC-backed DAO class...

private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);

}

public int countOfActorsByFirstName(String firstName) {

String sql = "select count(\*) from T\_ACTOR where first\_name = :first\_name";

Map<String, String> namedParameters = Collections.singletonMap("first\_name", firstName);

return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);

}

One nice feature related to the NamedParameterJdbcTemplate (and existing in the same Java package) is the SqlParameterSource interface. You have already seen an example of an implementation of this interface in one of the previous code snippet (the MapSqlParameterSource class). An SqlParameterSource is a source of named parameter values to a NamedParameterJdbcTemplate. The MapSqlParameterSource class is a very simple implementation that is simply an adapter around a java.util.Map, where the keys are the parameter names and the values are the parameter values.

Another SqlParameterSource implementation is the BeanPropertySqlParameterSource class. This class wraps an arbitrary JavaBean (that is, an instance of a class that adheres to the JavaBean conventions), and uses the properties of the wrapped JavaBean as the source of named parameter values.

public class Actor {

private Long id;

private String firstName;

private String lastName;

public String getFirstName() {

return this.firstName;

}

public String getLastName() {

return this.lastName;

}

public Long getId() {

return this.id;

}

// setters omitted...

}

// some JDBC-backed DAO class...

private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);

}

public int countOfActors(Actor exampleActor) {

// notice how the named parameters match the properties of the above 'Actor' class

String sql = "select count(\*) from T\_ACTOR where first\_name = :firstName and last\_name = :lastName";

SqlParameterSource namedParameters = new BeanPropertySqlParameterSource(exampleActor);

return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);

}

Remember that the NamedParameterJdbcTemplate class wraps a classic JdbcTemplate template; if you need access to the wrapped JdbcTemplate instance to access functionality only present in the JdbcTemplate class, you can use the getJdbcOperations() method to access the wrapped JdbcTemplate through the JdbcOperations interface.

See also JdbcTemplate best practices for guidelines on using the NamedParameterJdbcTemplate class in the context of an application.

3.2.3. SQLExceptionTranslator

SQLExceptionTranslator is an interface to be implemented by classes that can translate between SQLExceptions and Spring’s own org.springframework.dao.DataAccessException, which is agnostic in regard to data access strategy. Implementations can be generic (for example, using SQLState codes for JDBC) or proprietary (for example, using Oracle error codes) for greater precision.

SQLErrorCodeSQLExceptionTranslator is the implementation of SQLExceptionTranslator that is used by default. This implementation uses specific vendor codes. It is more precise than the SQLState implementation. The error code translations are based on codes held in a JavaBean type class called SQLErrorCodes. This class is created and populated by an SQLErrorCodesFactory which as the name suggests is a factory for creating SQLErrorCodes based on the contents of a configuration file named sql-error-codes.xml. This file is populated with vendor codes and based on the DatabaseProductName taken from the DatabaseMetaData. The codes for the actual database you are using are used.

The SQLErrorCodeSQLExceptionTranslator applies matching rules in the following sequence:

The SQLErrorCodesFactory is used by default to define Error codes and custom exception translations. They are looked up in a file named sql-error-codes.xml from the classpath and the matching SQLErrorCodes instance is located based on the database name from the database metadata of the database in use.

Any custom translation implemented by a subclass. Normally the provided concrete SQLErrorCodeSQLExceptionTranslator is used so this rule does not apply. It only applies if you have actually provided a subclass implementation.

Any custom implementation of the SQLExceptionTranslator interface that is provided as the customSqlExceptionTranslator property of the SQLErrorCodes class.

The list of instances of the CustomSQLErrorCodesTranslation class, provided for the customTranslations property of the SQLErrorCodes class, are searched for a match.

Error code matching is applied.

Use the fallback translator. SQLExceptionSubclassTranslator is the default fallback translator. If this translation is not available then the next fallback translator is the SQLStateSQLExceptionTranslator.

You can extend SQLErrorCodeSQLExceptionTranslator:

public class CustomSQLErrorCodesTranslator extends SQLErrorCodeSQLExceptionTranslator {

protected DataAccessException customTranslate(String task, String sql, SQLException sqlex) {

if (sqlex.getErrorCode() == -12345) {

return new DeadlockLoserDataAccessException(task, sqlex);

}

return null;

}

}

In this example, the specific error code -12345 is translated and other errors are left to be translated by the default translator implementation. To use this custom translator, it is necessary to pass it to the JdbcTemplate through the method setExceptionTranslator and to use this JdbcTemplate for all of the data access processing where this translator is needed. Here is an example of how this custom translator can be used:

private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

// create a JdbcTemplate and set data source

this.jdbcTemplate = new JdbcTemplate();

this.jdbcTemplate.setDataSource(dataSource);

// create a custom translator and set the DataSource for the default translation lookup

CustomSQLErrorCodesTranslator tr = new CustomSQLErrorCodesTranslator();

tr.setDataSource(dataSource);

this.jdbcTemplate.setExceptionTranslator(tr);

}

public void updateShippingCharge(long orderId, long pct) {

// use the prepared JdbcTemplate for this update

this.jdbcTemplate.update("update orders" +

" set shipping\_charge = shipping\_charge \* ? / 100" +

" where id = ?", pct, orderId);

}

The custom translator is passed a data source in order to look up the error codes in sql-error-codes.xml.

3.2.4. Executing statements

Executing an SQL statement requires very little code. You need a DataSource and a JdbcTemplate, including the convenience methods that are provided with the JdbcTemplate. The following example shows what you need to include for a minimal but fully functional class that creates a new table:

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAStatement {

private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

public void doExecute() {

this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");

}

}

3.2.5. Running queries

Some query methods return a single value. To retrieve a count or a specific value from one row, use queryForObject(..). The latter converts the returned JDBC Type to the Java class that is passed in as an argument. If the type conversion is invalid, then an InvalidDataAccessApiUsageException is thrown. Here is an example that contains two query methods, one for an int and one that queries for a String.

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class RunAQuery {

private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

public int getCount() {

return this.jdbcTemplate.queryForObject("select count(\*) from mytable", Integer.class);

}

public String getName() {

return this.jdbcTemplate.queryForObject("select name from mytable", String.class);

}

}

In addition to the single result query methods, several methods return a list with an entry for each row that the query returned. The most generic method is queryForList(..) which returns a List where each entry is a Map with each entry in the map representing the column value for that row. If you add a method to the above example to retrieve a list of all the rows, it would look like this:

private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

public List<Map<String, Object>> getList() {

return this.jdbcTemplate.queryForList("select \* from mytable");

}

The list returned would look something like this:

[{name=Bob, id=1}, {name=Mary, id=2}]

3.2.6. Updating the database

The following example shows a column updated for a certain primary key. In this example, an SQL statement has placeholders for row parameters. The parameter values can be passed in as varargs or alternatively as an array of objects. Thus primitives should be wrapped in the primitive wrapper classes explicitly or using auto-boxing.

import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAnUpdate {

private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

public void setName(int id, String name) {

this.jdbcTemplate.update("update mytable set name = ? where id = ?", name, id);

}

}

3.2.7. Retrieving auto-generated keys

An update() convenience method supports the retrieval of primary keys generated by the database. This support is part of the JDBC 3.0 standard; see Chapter 13.6 of the specification for details. The method takes a PreparedStatementCreator as its first argument, and this is the way the required insert statement is specified. The other argument is a KeyHolder, which contains the generated key on successful return from the update. There is not a standard single way to create an appropriate PreparedStatement (which explains why the method signature is the way it is). The following example works on Oracle but may not work on other platforms:

final String INSERT\_SQL = "insert into my\_test (name) values(?)";

final String name = "Rob";

KeyHolder keyHolder = new GeneratedKeyHolder();

jdbcTemplate.update(

new PreparedStatementCreator() {

public PreparedStatement createPreparedStatement(Connection connection) throws SQLException {

PreparedStatement ps = connection.prepareStatement(INSERT\_SQL, new String[] {"id"});

ps.setString(1, name);

return ps;

}

},

keyHolder);

// keyHolder.getKey() now contains the generated key

3.3. Controlling database connections

3.3.1. DataSource

Spring obtains a connection to the database through a DataSource. A DataSource is part of the JDBC specification and is a generalized connection factory. It allows a container or a framework to hide connection pooling and transaction management issues from the application code. As a developer, you need not know details about how to connect to the database; that is the responsibility of the administrator that sets up the datasource. You most likely fill both roles as you develop and test code, but you do not necessarily have to know how the production data source is configured.

When using Spring’s JDBC layer, you obtain a data source from JNDI or you configure your own with a connection pool implementation provided by a third party. Popular implementations are Apache Jakarta Commons DBCP and C3P0. Implementations in the Spring distribution are meant only for testing purposes and do not provide pooling.

This section uses Spring’s DriverManagerDataSource implementation, and several additional implementations are covered later.

Only use the DriverManagerDataSource class should only be used for testing purposes since it does not provide pooling and will perform poorly when multiple requests for a connection are made.

You obtain a connection with DriverManagerDataSource as you typically obtain a JDBC connection. Specify the fully qualified classname of the JDBC driver so that the DriverManager can load the driver class. Next, provide a URL that varies between JDBC drivers. (Consult the documentation for your driver for the correct value.) Then provide a username and a password to connect to the database. Here is an example of how to configure a DriverManagerDataSource in Java code:

DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName("org.hsqldb.jdbcDriver");

dataSource.setUrl("jdbc:hsqldb:hsql://localhost:");

dataSource.setUsername("sa");

dataSource.setPassword("");

Here is the corresponding XML configuration:

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="${jdbc.driverClassName}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

<context:property-placeholder location="jdbc.properties"/>

The following examples show the basic connectivity and configuration for DBCP and C3P0. To learn about more options that help control the pooling features, see the product documentation for the respective connection pooling implementations.

DBCP configuration:

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="${jdbc.driverClassName}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

<context:property-placeholder location="jdbc.properties"/>

C3P0 configuration:

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">

<property name="driverClass" value="${jdbc.driverClassName}"/>

<property name="jdbcUrl" value="${jdbc.url}"/>

<property name="user" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

<context:property-placeholder location="jdbc.properties"/>

3.3.2. DataSourceUtils

The DataSourceUtils class is a convenient and powerful helper class that provides static methods to obtain connections from JNDI and close connections if necessary. It supports thread-bound connections with, for example, DataSourceTransactionManager.

3.3.3. SmartDataSource

The SmartDataSource interface should be implemented by classes that can provide a connection to a relational database. It extends the DataSource interface to allow classes using it to query whether the connection should be closed after a given operation. This usage is efficient when you know that you will reuse a connection.

3.3.4. AbstractDataSource

AbstractDataSource is an abstract base class for Spring’s DataSource implementations that implements code that is common to all DataSource implementations. You extend the AbstractDataSource class if you are writing your own DataSource implementation.

3.3.5. SingleConnectionDataSource

The SingleConnectionDataSource class is an implementation of the SmartDataSource interface that wraps a single Connection that is not closed after each use. Obviously, this is not multi-threading capable.

If any client code calls close in the assumption of a pooled connection, as when using persistence tools, set the suppressClose property to true. This setting returns a close-suppressing proxy wrapping the physical connection. Be aware that you will not be able to cast this to a native Oracle Connection or the like anymore.

This is primarily a test class. For example, it enables easy testing of code outside an application server, in conjunction with a simple JNDI environment. In contrast to DriverManagerDataSource, it reuses the same connection all the time, avoiding excessive creation of physical connections.

3.3.6. DriverManagerDataSource

The DriverManagerDataSource class is an implementation of the standard DataSource interface that configures a plain JDBC driver through bean properties, and returns a new Connection every time.

This implementation is useful for test and stand-alone environments outside of a Java EE container, either as a DataSource bean in a Spring IoC container, or in conjunction with a simple JNDI environment. Pool-assuming Connection.close() calls will simply close the connection, so any DataSource-aware persistence code should work. However, using JavaBean-style connection pools such as commons-dbcp is so easy, even in a test environment, that it is almost always preferable to use such a connection pool over DriverManagerDataSource.

3.3.7. TransactionAwareDataSourceProxy

TransactionAwareDataSourceProxy is a proxy for a target DataSource, which wraps that target DataSource to add awareness of Spring-managed transactions. In this respect, it is similar to a transactional JNDI DataSource as provided by a Java EE server.

It is rarely desirable to use this class, except when already existing code that must be called and passed a standard JDBC DataSource interface implementation. In this case, it’s possible to still have this code be usable, and at the same time have this code participating in Spring managed transactions. It is generally preferable to write your own new code using the higher level abstractions for resource management, such as JdbcTemplate or DataSourceUtils.

(See the TransactionAwareDataSourceProxy javadocs for more details.)

3.3.8. DataSourceTransactionManager

The DataSourceTransactionManager class is a PlatformTransactionManager implementation for single JDBC datasources. It binds a JDBC connection from the specified data source to the currently executing thread, potentially allowing for one thread connection per data source.

Application code is required to retrieve the JDBC connection through DataSourceUtils.getConnection(DataSource) instead of Java EE’s standard DataSource.getConnection. It throws unchecked org.springframework.dao exceptions instead of checked SQLExceptions. All framework classes like JdbcTemplate use this strategy implicitly. If not used with this transaction manager, the lookup strategy behaves exactly like the common one - it can thus be used in any case.

The DataSourceTransactionManager class supports custom isolation levels, and timeouts that get applied as appropriate JDBC statement query timeouts. To support the latter, application code must either use JdbcTemplate or call the DataSourceUtils.applyTransactionTimeout(..) method for each created statement.

This implementation can be used instead of JtaTransactionManager in the single resource case, as it does not require the container to support JTA. Switching between both is just a matter of configuration, if you stick to the required connection lookup pattern. JTA does not support custom isolation levels!

3.4. JDBC batch operations

Most JDBC drivers provide improved performance if you batch multiple calls to the same prepared statement. By grouping updates into batches you limit the number of round trips to the database.

3.4.1. Basic batch operations with the JdbcTemplate

You accomplish JdbcTemplate batch processing by implementing two methods of a special interface, BatchPreparedStatementSetter, and passing that in as the second parameter in your batchUpdate method call. Use the getBatchSize method to provide the size of the current batch. Use the setValues method to set the values for the parameters of the prepared statement. This method will be called the number of times that you specified in the getBatchSize call. The following example updates the actor table based on entries in a list. The entire list is used as the batch in this example:

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

public int[] batchUpdate(final List<Actor> actors) {

return this.jdbcTemplate.batchUpdate(

"update t\_actor set first\_name = ?, last\_name = ? where id = ?",

new BatchPreparedStatementSetter() {

public void setValues(PreparedStatement ps, int i) throws SQLException {

ps.setString(1, actors.get(i).getFirstName());

ps.setString(2, actors.get(i).getLastName());

ps.setLong(3, actors.get(i).getId().longValue());

}

public int getBatchSize() {

return actors.size();

}

});

}

// ... additional methods

}

If you are processing a stream of updates or reading from a file, then you might have a preferred batch size, but the last batch might not have that number of entries. In this case you can use the InterruptibleBatchPreparedStatementSetter interface, which allows you to interrupt a batch once the input source is exhausted. The isBatchExhausted method allows you to signal the end of the batch.

3.4.2. Batch operations with a List of objects

Both the JdbcTemplate and the NamedParameterJdbcTemplate provides an alternate way of providing the batch update. Instead of implementing a special batch interface, you provide all parameter values in the call as a list. The framework loops over these values and uses an internal prepared statement setter. The API varies depending on whether you use named parameters. For the named parameters you provide an array of SqlParameterSource, one entry for each member of the batch. You can use the SqlParameterSourceUtils.createBatch convenience methods to create this array, passing in an array of bean-style objects (with getter methods corresponding to parameters) and/or String-keyed Maps (containing the corresponding parameters as values).

This example shows a batch update using named parameters:

public class JdbcActorDao implements ActorDao {

private NamedParameterTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);

}

public int[] batchUpdate(List<Actor> actors) {

return this.namedParameterJdbcTemplate.batchUpdate(

"update t\_actor set first\_name = :firstName, last\_name = :lastName where id = :id",

SqlParameterSourceUtils.createBatch(actors));

}

// ... additional methods

}

For an SQL statement using the classic "?" placeholders, you pass in a list containing an object array with the update values. This object array must have one entry for each placeholder in the SQL statement, and they must be in the same order as they are defined in the SQL statement.

The same example using classic JDBC "?" placeholders:

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

public int[] batchUpdate(final List<Actor> actors) {

List<Object[]> batch = new ArrayList<Object[]>();

for (Actor actor : actors) {

Object[] values = new Object[] {

actor.getFirstName(), actor.getLastName(), actor.getId()};

batch.add(values);

}

return this.jdbcTemplate.batchUpdate(

"update t\_actor set first\_name = ?, last\_name = ? where id = ?",

batch);

}

// ... additional methods

}

All of the above batch update methods return an int array containing the number of affected rows for each batch entry. This count is reported by the JDBC driver. If the count is not available, the JDBC driver returns a -2 value.

In such a scenario with automatic setting of values on an underlying PreparedStatement, the corresponding JDBC type for each value needs to be derived from the given Java type. While this usually works well, there is a potential for issues, e.g. with Map-contained null values: Spring will by default call ParameterMetaData.getParameterType in such a case which may be expensive with your JDBC driver. Please make sure to use a recent driver version, and consider setting the "spring.jdbc.getParameterType.ignore" property to "true" (as a JVM system property or in a spring.properties file in the root of your classpath) if you encounter a performance issue, e.g. as reported on Oracle 12c (SPR-16139).

Alternatively, simply consider specifying the corresponding JDBC types explicitly: either via a 'BatchPreparedStatementSetter' as shown above, or via an explicit type array given to a 'List<Object[]>' based call, or via 'registerSqlType' calls on a custom 'MapSqlParameterSource' instance, or via a 'BeanPropertySqlParameterSource' which derives the SQL type from the Java-declared property type even for a null value.

3.4.3. Batch operations with multiple batches

The last example of a batch update deals with batches that are so large that you want to break them up into several smaller batches. You can of course do this with the methods mentioned above by making multiple calls to the batchUpdate method, but there is now a more convenient method. This method takes, in addition to the SQL statement, a Collection of objects containing the parameters, the number of updates to make for each batch and a ParameterizedPreparedStatementSetter to set the values for the parameters of the prepared statement. The framework loops over the provided values and breaks the update calls into batches of the size specified.

This example shows a batch update using a batch size of 100:

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

public int[][] batchUpdate(final Collection<Actor> actors) {

int[][] updateCounts = jdbcTemplate.batchUpdate(

"update t\_actor set first\_name = ?, last\_name = ? where id = ?",

actors,

100,

new ParameterizedPreparedStatementSetter<Actor>() {

public void setValues(PreparedStatement ps, Actor argument) throws SQLException {

ps.setString(1, argument.getFirstName());

ps.setString(2, argument.getLastName());

ps.setLong(3, argument.getId().longValue());

}

});

return updateCounts;

}

// ... additional methods

}

The batch update methods for this call returns an array of int arrays containing an array entry for each batch with an array of the number of affected rows for each update. The top level array’s length indicates the number of batches executed and the second level array’s length indicates the number of updates in that batch. The number of updates in each batch should be the batch size provided for all batches except for the last one that might be less, depending on the total number of update objects provided. The update count for each update statement is the one reported by the JDBC driver. If the count is not available, the JDBC driver returns a -2 value.

3.5. Simplifying JDBC operations with the SimpleJdbc classes

The SimpleJdbcInsert and SimpleJdbcCall classes provide a simplified configuration by taking advantage of database metadata that can be retrieved through the JDBC driver. This means there is less to configure up front, although you can override or turn off the metadata processing if you prefer to provide all the details in your code.

3.5.1. Inserting data using SimpleJdbcInsert

Let’s start by looking at the SimpleJdbcInsert class with the minimal amount of configuration options. You should instantiate the SimpleJdbcInsert in the data access layer’s initialization method. For this example, the initializing method is the setDataSource method. You do not need to subclass the SimpleJdbcInsert class; simply create a new instance and set the table name using the withTableName method. Configuration methods for this class follow the "fluid" style that returns the instance of the SimpleJdbcInsert, which allows you to chain all configuration methods. This example uses only one configuration method; you will see examples of multiple ones later.

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

private SimpleJdbcInsert insertActor;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

this.insertActor = new SimpleJdbcInsert(dataSource).withTableName("t\_actor");

}

public void add(Actor actor) {

Map<String, Object> parameters = new HashMap<String, Object>(3);

parameters.put("id", actor.getId());

parameters.put("first\_name", actor.getFirstName());

parameters.put("last\_name", actor.getLastName());

insertActor.execute(parameters);

}

// ... additional methods

}

The execute method used here takes a plain java.utils.Map as its only parameter. The important thing to note here is that the keys used for the Map must match the column names of the table as defined in the database. This is because we read the metadata in order to construct the actual insert statement.

3.5.2. Retrieving auto-generated keys using SimpleJdbcInsert

This example uses the same insert as the preceding, but instead of passing in the id it retrieves the auto-generated key and sets it on the new Actor object. When you create the SimpleJdbcInsert, in addition to specifying the table name, you specify the name of the generated key column with the usingGeneratedKeyColumns method.

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

private SimpleJdbcInsert insertActor;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

this.insertActor = new SimpleJdbcInsert(dataSource)

.withTableName("t\_actor")

.usingGeneratedKeyColumns("id");

}

public void add(Actor actor) {

Map<String, Object> parameters = new HashMap<String, Object>(2);

parameters.put("first\_name", actor.getFirstName());

parameters.put("last\_name", actor.getLastName());

Number newId = insertActor.executeAndReturnKey(parameters);

actor.setId(newId.longValue());

}

// ... additional methods

}

The main difference when executing the insert by this second approach is that you do not add the id to the Map and you call the executeAndReturnKey method. This returns a java.lang.Number object with which you can create an instance of the numerical type that is used in our domain class. You cannot rely on all databases to return a specific Java class here; java.lang.Number is the base class that you can rely on. If you have multiple auto-generated columns, or the generated values are non-numeric, then you can use a KeyHolder that is returned from the executeAndReturnKeyHolder method.

3.5.3. Specifying columns for a SimpleJdbcInsert

You can limit the columns for an insert by specifying a list of column names with the usingColumns method:

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

private SimpleJdbcInsert insertActor;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

this.insertActor = new SimpleJdbcInsert(dataSource)

.withTableName("t\_actor")

.usingColumns("first\_name", "last\_name")

.usingGeneratedKeyColumns("id");

}

public void add(Actor actor) {

Map<String, Object> parameters = new HashMap<String, Object>(2);

parameters.put("first\_name", actor.getFirstName());

parameters.put("last\_name", actor.getLastName());

Number newId = insertActor.executeAndReturnKey(parameters);

actor.setId(newId.longValue());

}

// ... additional methods

}

The execution of the insert is the same as if you had relied on the metadata to determine which columns to use.

3.5.4. Using SqlParameterSource to provide parameter values

Using a Map to provide parameter values works fine, but it’s not the most convenient class to use. Spring provides a couple of implementations of the SqlParameterSource interface that can be used instead.The first one is BeanPropertySqlParameterSource, which is a very convenient class if you have a JavaBean-compliant class that contains your values. It will use the corresponding getter method to extract the parameter values. Here is an example:

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

private SimpleJdbcInsert insertActor;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

this.insertActor = new SimpleJdbcInsert(dataSource)

.withTableName("t\_actor")

.usingGeneratedKeyColumns("id");

}

public void add(Actor actor) {

SqlParameterSource parameters = new BeanPropertySqlParameterSource(actor);

Number newId = insertActor.executeAndReturnKey(parameters);

actor.setId(newId.longValue());

}

// ... additional methods

}

Another option is the MapSqlParameterSource that resembles a Map but provides a more convenient addValue method that can be chained.

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

private SimpleJdbcInsert insertActor;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

this.insertActor = new SimpleJdbcInsert(dataSource)

.withTableName("t\_actor")

.usingGeneratedKeyColumns("id");

}

public void add(Actor actor) {

SqlParameterSource parameters = new MapSqlParameterSource()

.addValue("first\_name", actor.getFirstName())

.addValue("last\_name", actor.getLastName());

Number newId = insertActor.executeAndReturnKey(parameters);

actor.setId(newId.longValue());

}

// ... additional methods

}

As you can see, the configuration is the same; only the executing code has to change to use these alternative input classes.

3.5.5. Calling a stored procedure with SimpleJdbcCall

The SimpleJdbcCall class leverages metadata in the database to look up names of in and out parameters, so that you do not have to declare them explicitly. You can declare parameters if you prefer to do that, or if you have parameters such as ARRAY or STRUCT that do not have an automatic mapping to a Java class. The first example shows a simple procedure that returns only scalar values in VARCHAR and DATE format from a MySQL database. The example procedure reads a specified actor entry and returns first\_name, last\_name, and birth\_date columns in the form of out parameters.

CREATE PROCEDURE read\_actor (

IN in\_id INTEGER,

OUT out\_first\_name VARCHAR(100),

OUT out\_last\_name VARCHAR(100),

OUT out\_birth\_date DATE)

BEGIN

SELECT first\_name, last\_name, birth\_date

INTO out\_first\_name, out\_last\_name, out\_birth\_date

FROM t\_actor where id = in\_id;

END;

The in\_id parameter contains the id of the actor you are looking up. The out parameters return the data read from the table.

The SimpleJdbcCall is declared in a similar manner to the SimpleJdbcInsert. You should instantiate and configure the class in the initialization method of your data access layer. Compared to the StoredProcedure class, you don’t have to create a subclass and you don’t have to declare parameters that can be looked up in the database metadata. Following is an example of a SimpleJdbcCall configuration using the above stored procedure. The only configuration option, in addition to the DataSource, is the name of the stored procedure.

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

private SimpleJdbcCall procReadActor;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

this.procReadActor = new SimpleJdbcCall(dataSource)

.withProcedureName("read\_actor");

}

public Actor readActor(Long id) {

SqlParameterSource in = new MapSqlParameterSource()

.addValue("in\_id", id);

Map out = procReadActor.execute(in);

Actor actor = new Actor();

actor.setId(id);

actor.setFirstName((String) out.get("out\_first\_name"));

actor.setLastName((String) out.get("out\_last\_name"));

actor.setBirthDate((Date) out.get("out\_birth\_date"));

return actor;

}

// ... additional methods

}

The code you write for the execution of the call involves creating an SqlParameterSource containing the IN parameter. It’s important to match the name provided for the input value with that of the parameter name declared in the stored procedure. The case does not have to match because you use metadata to determine how database objects should be referred to in a stored procedure. What is specified in the source for the stored procedure is not necessarily the way it is stored in the database. Some databases transform names to all upper case while others use lower case or use the case as specified.

The execute method takes the IN parameters and returns a Map containing any out parameters keyed by the name as specified in the stored procedure. In this case they are out\_first\_name, out\_last\_name and out\_birth\_date.

The last part of the execute method creates an Actor instance to use to return the data retrieved. Again, it is important to use the names of the out parameters as they are declared in the stored procedure. Also, the case in the names of the out parameters stored in the results map matches that of the out parameter names in the database, which could vary between databases. To make your code more portable you should do a case-insensitive lookup or instruct Spring to use a LinkedCaseInsensitiveMap. To do the latter, you create your own JdbcTemplate and set the setResultsMapCaseInsensitive property to true. Then you pass this customized JdbcTemplate instance into the constructor of your SimpleJdbcCall. Here is an example of this configuration:

public class JdbcActorDao implements ActorDao {

private SimpleJdbcCall procReadActor;

public void setDataSource(DataSource dataSource) {

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

jdbcTemplate.setResultsMapCaseInsensitive(true);

this.procReadActor = new SimpleJdbcCall(jdbcTemplate)

.withProcedureName("read\_actor");

}

// ... additional methods

}

By taking this action, you avoid conflicts in the case used for the names of your returned out parameters.

3.5.6. Explicitly declaring parameters to use for a SimpleJdbcCall

You have seen how the parameters are deduced based on metadata, but you can declare then explicitly if you wish. You do this by creating and configuring SimpleJdbcCall with the declareParameters method, which takes a variable number of SqlParameter objects as input. See the next section for details on how to define an SqlParameter.

Explicit declarations are necessary if the database you use is not a Spring-supported database. Currently Spring supports metadata lookup of stored procedure calls for the following databases: Apache Derby, DB2, MySQL, Microsoft SQL Server, Oracle, and Sybase. We also support metadata lookup of stored functions for MySQL, Microsoft SQL Server, and Oracle.

You can opt to declare one, some, or all the parameters explicitly. The parameter metadata is still used where you do not declare parameters explicitly. To bypass all processing of metadata lookups for potential parameters and only use the declared parameters, you call the method withoutProcedureColumnMetaDataAccess as part of the declaration. Suppose that you have two or more different call signatures declared for a database function. In this case you call the useInParameterNames to specify the list of IN parameter names to include for a given signature.

The following example shows a fully declared procedure call, using the information from the preceding example.

public class JdbcActorDao implements ActorDao {

private SimpleJdbcCall procReadActor;

public void setDataSource(DataSource dataSource) {

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

jdbcTemplate.setResultsMapCaseInsensitive(true);

this.procReadActor = new SimpleJdbcCall(jdbcTemplate)

.withProcedureName("read\_actor")

.withoutProcedureColumnMetaDataAccess()

.useInParameterNames("in\_id")

.declareParameters(

new SqlParameter("in\_id", Types.NUMERIC),

new SqlOutParameter("out\_first\_name", Types.VARCHAR),

new SqlOutParameter("out\_last\_name", Types.VARCHAR),

new SqlOutParameter("out\_birth\_date", Types.DATE)

);

}

// ... additional methods

}

The execution and end results of the two examples are the same; this one specifies all details explicitly rather than relying on metadata.

3.5.7. How to define SqlParameters

To define a parameter for the SimpleJdbc classes and also for the RDBMS operations classes, covered in Modeling JDBC operations as Java objects, you use an SqlParameter or one of its subclasses. You typically specify the parameter name and SQL type in the constructor. The SQL type is specified using the java.sql.Types constants. We have already seen declarations like:

new SqlParameter("in\_id", Types.NUMERIC),

new SqlOutParameter("out\_first\_name", Types.VARCHAR),

The first line with the SqlParameter declares an IN parameter. IN parameters can be used for both stored procedure calls and for queries using the SqlQuery and its subclasses covered in the following section.

The second line with the SqlOutParameter declares an out parameter to be used in a stored procedure call. There is also an SqlInOutParameter for InOut parameters, parameters that provide an IN value to the procedure and that also return a value.

Only parameters declared as SqlParameter and SqlInOutParameter will be used to provide input values. This is different from the StoredProcedure class, which for backwards compatibility reasons allows input values to be provided for parameters declared as SqlOutParameter.

For IN parameters, in addition to the name and the SQL type, you can specify a scale for numeric data or a type name for custom database types. For out parameters, you can provide a RowMapper to handle mapping of rows returned from a REF cursor. Another option is to specify an SqlReturnType that provides an opportunity to define customized handling of the return values.

3.5.8. Calling a stored function using SimpleJdbcCall

You call a stored function in almost the same way as you call a stored procedure, except that you provide a function name rather than a procedure name. You use the withFunctionName method as part of the configuration to indicate that we want to make a call to a function, and the corresponding string for a function call is generated. A specialized execute call, executeFunction, is used to execute the function and it returns the function return value as an object of a specified type, which means you do not have to retrieve the return value from the results map. A similar convenience method named executeObject is also available for stored procedures that only have one out parameter. The following example is based on a stored function named get\_actor\_name that returns an actor’s full name. Here is the MySQL source for this function:

CREATE FUNCTION get\_actor\_name (in\_id INTEGER)

RETURNS VARCHAR(200) READS SQL DATA

BEGIN

DECLARE out\_name VARCHAR(200);

SELECT concat(first\_name, ' ', last\_name)

INTO out\_name

FROM t\_actor where id = in\_id;

RETURN out\_name;

END;

To call this function we again create a SimpleJdbcCall in the initialization method.

public class JdbcActorDao implements ActorDao {

private JdbcTemplate jdbcTemplate;

private SimpleJdbcCall funcGetActorName;

public void setDataSource(DataSource dataSource) {

this.jdbcTemplate = new JdbcTemplate(dataSource);

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

jdbcTemplate.setResultsMapCaseInsensitive(true);

this.funcGetActorName = new SimpleJdbcCall(jdbcTemplate)

.withFunctionName("get\_actor\_name");

}

public String getActorName(Long id) {

SqlParameterSource in = new MapSqlParameterSource()

.addValue("in\_id", id);

String name = funcGetActorName.executeFunction(String.class, in);

return name;

}

// ... additional methods

}

The execute method used returns a String containing the return value from the function call.

3.5.9. Returning ResultSet/REF Cursor from a SimpleJdbcCall

Calling a stored procedure or function that returns a result set is a bit tricky. Some databases return result sets during the JDBC results processing while others require an explicitly registered out parameter of a specific type. Both approaches need additional processing to loop over the result set and process the returned rows. With the SimpleJdbcCall you use the returningResultSet method and declare a RowMapper implementation to be used for a specific parameter. In the case where the result set is returned during the results processing, there are no names defined, so the returned results will have to match the order in which you declare the RowMapper implementations. The name specified is still used to store the processed list of results in the results map that is returned from the execute statement.

The next example uses a stored procedure that takes no IN parameters and returns all rows from the t\_actor table. Here is the MySQL source for this procedure:

CREATE PROCEDURE read\_all\_actors()

BEGIN

SELECT a.id, a.first\_name, a.last\_name, a.birth\_date FROM t\_actor a;

END;

To call this procedure you declare the RowMapper. Because the class you want to map to follows the JavaBean rules, you can use a BeanPropertyRowMapper that is created by passing in the required class to map to in the newInstance method.

public class JdbcActorDao implements ActorDao {

private SimpleJdbcCall procReadAllActors;

public void setDataSource(DataSource dataSource) {

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

jdbcTemplate.setResultsMapCaseInsensitive(true);

this.procReadAllActors = new SimpleJdbcCall(jdbcTemplate)

.withProcedureName("read\_all\_actors")

.returningResultSet("actors",

BeanPropertyRowMapper.newInstance(Actor.class));

}

public List getActorsList() {

Map m = procReadAllActors.execute(new HashMap<String, Object>(0));

return (List) m.get("actors");

}

// ... additional methods

}

The execute call passes in an empty Map because this call does not take any parameters. The list of Actors is then retrieved from the results map and returned to the caller.

3.6. Modeling JDBC operations as Java objects

The org.springframework.jdbc.object package contains classes that allow you to access the database in a more object-oriented manner. As an example, you can execute queries and get the results back as a list containing business objects with the relational column data mapped to the properties of the business object. You can also execute stored procedures and run update, delete, and insert statements.

Many Spring developers believe that the various RDBMS operation classes described below (with the exception of the StoredProcedure class) can often be replaced with straight JdbcTemplate calls. Often it is simpler to write a DAO method that simply calls a method on a JdbcTemplate directly (as opposed to encapsulating a query as a full-blown class).

However, if you are getting measurable value from using the RDBMS operation classes, continue using these classes.

3.6.1. SqlQuery

SqlQuery is a reusable, threadsafe class that encapsulates an SQL query. Subclasses must implement the newRowMapper(..) method to provide a RowMapper instance that can create one object per row obtained from iterating over the ResultSet that is created during the execution of the query. The SqlQuery class is rarely used directly because the MappingSqlQuery subclass provides a much more convenient implementation for mapping rows to Java classes. Other implementations that extend SqlQuery are MappingSqlQueryWithParameters and UpdatableSqlQuery.

3.6.2. MappingSqlQuery

MappingSqlQuery is a reusable query in which concrete subclasses must implement the abstract mapRow(..) method to convert each row of the supplied ResultSet into an object of the type specified. The following example shows a custom query that maps the data from the t\_actor relation to an instance of the Actor class.

public class ActorMappingQuery extends MappingSqlQuery<Actor> {

public ActorMappingQuery(DataSource ds) {

super(ds, "select id, first\_name, last\_name from t\_actor where id = ?");

declareParameter(new SqlParameter("id", Types.INTEGER));

compile();

}

@Override

protected Actor mapRow(ResultSet rs, int rowNumber) throws SQLException {

Actor actor = new Actor();

actor.setId(rs.getLong("id"));

actor.setFirstName(rs.getString("first\_name"));

actor.setLastName(rs.getString("last\_name"));

return actor;

}

}

The class extends MappingSqlQuery parameterized with the Actor type. The constructor for this customer query takes the DataSource as the only parameter. In this constructor you call the constructor on the superclass with the DataSource and the SQL that should be executed to retrieve the rows for this query. This SQL will be used to create a PreparedStatement so it may contain place holders for any parameters to be passed in during execution. You must declare each parameter using the declareParameter method passing in an SqlParameter. The SqlParameter takes a name and the JDBC type as defined in java.sql.Types. After you define all parameters, you call the compile() method so the statement can be prepared and later executed. This class is thread-safe after it is compiled, so as long as these instances are created when the DAO is initialized they can be kept as instance variables and be reused.

private ActorMappingQuery actorMappingQuery;

@Autowired

public void setDataSource(DataSource dataSource) {

this.actorMappingQuery = new ActorMappingQuery(dataSource);

}

public Customer getCustomer(Long id) {

return actorMappingQuery.findObject(id);

}

The method in this example retrieves the customer with the id that is passed in as the only parameter. Since we only want one object returned we simply call the convenience method findObject with the id as parameter. If we had instead a query that returned a list of objects and took additional parameters then we would use one of the execute methods that takes an array of parameter values passed in as varargs.

public List<Actor> searchForActors(int age, String namePattern) {

List<Actor> actors = actorSearchMappingQuery.execute(age, namePattern);

return actors;

}

3.6.3. SqlUpdate

The SqlUpdate class encapsulates an SQL update. Like a query, an update object is reusable, and like all RdbmsOperation classes, an update can have parameters and is defined in SQL. This class provides a number of update(..) methods analogous to the execute(..) methods of query objects. The SQLUpdate class is concrete. It can be subclassed, for example, to add a custom update method, as in the following snippet where it’s simply called execute. However, you don’t have to subclass the SqlUpdate class since it can easily be parameterized by setting SQL and declaring parameters.

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;

import org.springframework.jdbc.object.SqlUpdate;

public class UpdateCreditRating extends SqlUpdate {

public UpdateCreditRating(DataSource ds) {

setDataSource(ds);

setSql("update customer set credit\_rating = ? where id = ?");

declareParameter(new SqlParameter("creditRating", Types.NUMERIC));

declareParameter(new SqlParameter("id", Types.NUMERIC));

compile();

}

/\*\*

\* @param id for the Customer to be updated

\* @param rating the new value for credit rating

\* @return number of rows updated

\*/

public int execute(int id, int rating) {

return update(rating, id);

}

}

3.6.4. StoredProcedure

The StoredProcedure class is a superclass for object abstractions of RDBMS stored procedures. This class is abstract, and its various execute(..) methods have protected access, preventing use other than through a subclass that offers tighter typing.

The inherited sql property will be the name of the stored procedure in the RDBMS.

To define a parameter for the StoredProcedure class, you use an SqlParameter or one of its subclasses. You must specify the parameter name and SQL type in the constructor like in the following code snippet. The SQL type is specified using the java.sql.Types constants.

new SqlParameter("in\_id", Types.NUMERIC),

new SqlOutParameter("out\_first\_name", Types.VARCHAR),

The first line with the SqlParameter declares an IN parameter. IN parameters can be used for both stored procedure calls and for queries using the SqlQuery and its subclasses covered in the following section.

The second line with the SqlOutParameter declares an out parameter to be used in the stored procedure call. There is also an SqlInOutParameter for InOut parameters, parameters that provide an in value to the procedure and that also return a value.

For in parameters, in addition to the name and the SQL type, you can specify a scale for numeric data or a type name for custom database types. For out parameters you can provide a RowMapper to handle mapping of rows returned from a REF cursor. Another option is to specify an SqlReturnType that enables you to define customized handling of the return values.

Here is an example of a simple DAO that uses a StoredProcedure to call a function, sysdate(),which comes with any Oracle database. To use the stored procedure functionality you have to create a class that extends StoredProcedure. In this example, the StoredProcedure class is an inner class, but if you need to reuse the StoredProcedure you declare it as a top-level class. This example has no input parameters, but an output parameter is declared as a date type using the class SqlOutParameter. The execute() method executes the procedure and extracts the returned date from the results Map. The results Map has an entry for each declared output parameter, in this case only one, using the parameter name as the key.

import java.sql.Types;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.SqlOutParameter;

import org.springframework.jdbc.object.StoredProcedure;

public class StoredProcedureDao {

private GetSysdateProcedure getSysdate;

@Autowired

public void init(DataSource dataSource) {

this.getSysdate = new GetSysdateProcedure(dataSource);

}

public Date getSysdate() {

return getSysdate.execute();

}

private class GetSysdateProcedure extends StoredProcedure {

private static final String SQL = "sysdate";

public GetSysdateProcedure(DataSource dataSource) {

setDataSource(dataSource);

setFunction(true);

setSql(SQL);

declareParameter(new SqlOutParameter("date", Types.DATE));

compile();

}

public Date execute() {

// the 'sysdate' sproc has no input parameters, so an empty Map is supplied...

Map<String, Object> results = execute(new HashMap<String, Object>());

Date sysdate = (Date) results.get("date");

return sysdate;

}

}

}

The following example of a StoredProcedure has two output parameters (in this case, Oracle REF cursors).

import java.util.HashMap;

import java.util.Map;

import javax.sql.DataSource;

import oracle.jdbc.OracleTypes;

import org.springframework.jdbc.core.SqlOutParameter;

import org.springframework.jdbc.object.StoredProcedure;

public class TitlesAndGenresStoredProcedure extends StoredProcedure {

private static final String SPROC\_NAME = "AllTitlesAndGenres";

public TitlesAndGenresStoredProcedure(DataSource dataSource) {

super(dataSource, SPROC\_NAME);

declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new TitleMapper()));

declareParameter(new SqlOutParameter("genres", OracleTypes.CURSOR, new GenreMapper()));

compile();

}

public Map<String, Object> execute() {

// again, this sproc has no input parameters, so an empty Map is supplied

return super.execute(new HashMap<String, Object>());

}

}

Notice how the overloaded variants of the declareParameter(..) method that have been used in the TitlesAndGenresStoredProcedure constructor are passed RowMapper implementation instances; this is a very convenient and powerful way to reuse existing functionality. The code for the two RowMapper implementations is provided below.

The TitleMapper class maps a ResultSet to a Title domain object for each row in the supplied ResultSet:

import java.sql.ResultSet;

import java.sql.SQLException;

import com.foo.domain.Title;

import org.springframework.jdbc.core.RowMapper;

public final class TitleMapper implements RowMapper<Title> {

public Title mapRow(ResultSet rs, int rowNum) throws SQLException {

Title title = new Title();

title.setId(rs.getLong("id"));

title.setName(rs.getString("name"));

return title;

}

}

The GenreMapper class maps a ResultSet to a Genre domain object for each row in the supplied ResultSet.

import java.sql.ResultSet;

import java.sql.SQLException;

import com.foo.domain.Genre;

import org.springframework.jdbc.core.RowMapper;

public final class GenreMapper implements RowMapper<Genre> {

public Genre mapRow(ResultSet rs, int rowNum) throws SQLException {

return new Genre(rs.getString("name"));

}

}

To pass parameters to a stored procedure that has one or more input parameters in its definition in the RDBMS, you can code a strongly typed execute(..) method that would delegate to the untyped execute(Map) method in the superclass; for example:

import java.sql.Types;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import javax.sql.DataSource;

import oracle.jdbc.OracleTypes;

import org.springframework.jdbc.core.SqlOutParameter;

import org.springframework.jdbc.core.SqlParameter;

import org.springframework.jdbc.object.StoredProcedure;

public class TitlesAfterDateStoredProcedure extends StoredProcedure {

private static final String SPROC\_NAME = "TitlesAfterDate";

private static final String CUTOFF\_DATE\_PARAM = "cutoffDate";

public TitlesAfterDateStoredProcedure(DataSource dataSource) {

super(dataSource, SPROC\_NAME);

declareParameter(new SqlParameter(CUTOFF\_DATE\_PARAM, Types.DATE);

declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new TitleMapper()));

compile();

}

public Map<String, Object> execute(Date cutoffDate) {

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

inputs.put(CUTOFF\_DATE\_PARAM, cutoffDate);

return super.execute(inputs);

}

}

3.7. Common problems with parameter and data value handling

Common problems with parameters and data values exist in the different approaches provided by Spring Framework’s JDBC support.

3.7.1. Providing SQL type information for parameters

Usually Spring determines the SQL type of the parameters based on the type of parameter passed in. It is possible to explicitly provide the SQL type to be used when setting parameter values. This is sometimes necessary to correctly set NULL values.

You can provide SQL type information in several ways:

Many update and query methods of the JdbcTemplate take an additional parameter in the form of an int array. This array is used to indicate the SQL type of the corresponding parameter using constant values from the java.sql.Types class. Provide one entry for each parameter.

You can use the SqlParameterValue class to wrap the parameter value that needs this additional information.Create a new instance for each value and pass in the SQL type and parameter value in the constructor. You can also provide an optional scale parameter for numeric values.

For methods working with named parameters, use the SqlParameterSource classes BeanPropertySqlParameterSource or MapSqlParameterSource. They both have methods for registering the SQL type for any of the named parameter values.

3.7.2. Handling BLOB and CLOB objects

You can store images, other binary data, and large chunks of text in the database. These large objects are called BLOBs (Binary Large OBject) for binary data and CLOBs (Character Large OBject) for character data. In Spring you can handle these large objects by using the JdbcTemplate directly and also when using the higher abstractions provided by RDBMS Objects and the SimpleJdbc classes. All of these approaches use an implementation of the LobHandler interface for the actual management of the LOB (Large OBject) data. The LobHandler provides access to a LobCreator class, through the getLobCreator method, used for creating new LOB objects to be inserted.

The LobCreator/LobHandler provides the following support for LOB input and output:

BLOB

byte[] — getBlobAsBytes and setBlobAsBytes

InputStream — getBlobAsBinaryStream and setBlobAsBinaryStream

CLOB

String — getClobAsString and setClobAsString

InputStream — getClobAsAsciiStream and setClobAsAsciiStream

Reader — getClobAsCharacterStream and setClobAsCharacterStream

The next example shows how to create and insert a BLOB. Later you will see how to read it back from the database.

This example uses a JdbcTemplate and an implementation of the AbstractLobCreatingPreparedStatementCallback. It implements one method, setValues. This method provides a LobCreator that you use to set the values for the LOB columns in your SQL insert statement.

For this example we assume that there is a variable, lobHandler, that already is set to an instance of a DefaultLobHandler. You typically set this value through dependency injection.

final File blobIn = new File("spring2004.jpg");

final InputStream blobIs = new FileInputStream(blobIn);

final File clobIn = new File("large.txt");

final InputStream clobIs = new FileInputStream(clobIn);

final InputStreamReader clobReader = new InputStreamReader(clobIs);

jdbcTemplate.execute(

"INSERT INTO lob\_table (id, a\_clob, a\_blob) VALUES (?, ?, ?)",

new AbstractLobCreatingPreparedStatementCallback(lobHandler) {

protected void setValues(PreparedStatement ps, LobCreator lobCreator) throws SQLException {

ps.setLong(1, 1L);

lobCreator.setClobAsCharacterStream(ps, 2, clobReader, (int)clobIn.length());

lobCreator.setBlobAsBinaryStream(ps, 3, blobIs, (int)blobIn.length());

}

}

);

blobIs.close();

clobReader.close();

Pass in the lobHandler that in this example is a plain DefaultLobHandler.

Using the method setClobAsCharacterStream, pass in the contents of the CLOB.

Using the method setBlobAsBinaryStream, pass in the contents of the BLOB.

If you invoke the setBlobAsBinaryStream, setClobAsAsciiStream, or setClobAsCharacterStream method on the LobCreator returned from DefaultLobHandler.getLobCreator(), you can optionally specify a negative value for the contentLength argument. If the specified content length is negative, the DefaultLobHandler will use the JDBC 4.0 variants of the set-stream methods without a length parameter; otherwise, it will pass the specified length on to the driver.

Consult the documentation for the JDBC driver in use to verify support for streaming a LOB without providing the content length.

Now it’s time to read the LOB data from the database. Again, you use a JdbcTemplate with the same instance variable lobHandler and a reference to a DefaultLobHandler.

List<Map<String, Object>> l = jdbcTemplate.query("select id, a\_clob, a\_blob from lob\_table",

new RowMapper<Map<String, Object>>() {

public Map<String, Object> mapRow(ResultSet rs, int i) throws SQLException {

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

String clobText = lobHandler.getClobAsString(rs, "a\_clob");

results.put("CLOB", clobText);

byte[] blobBytes = lobHandler.getBlobAsBytes(rs, "a\_blob");

results.put("BLOB", blobBytes);

return results;

}

});

Using the method getClobAsString, retrieve the contents of the CLOB.

Using the method getBlobAsBytes, retrieve the contents of the BLOB.

3.7.3. Passing in lists of values for IN clause

The SQL standard allows for selecting rows based on an expression that includes a variable list of values. A typical example would be select \* from T\_ACTOR where id in (1, 2, 3). This variable list is not directly supported for prepared statements by the JDBC standard; you cannot declare a variable number of placeholders. You need a number of variations with the desired number of placeholders prepared, or you need to generate the SQL string dynamically once you know how many placeholders are required. The named parameter support provided in the NamedParameterJdbcTemplate and JdbcTemplate takes the latter approach. Pass in the values as a java.util.List of primitive objects. This list will be used to insert the required placeholders and pass in the values during the statement execution.

Be careful when passing in many values. The JDBC standard does not guarantee that you can use more than 100 values for an in expression list. Various databases exceed this number, but they usually have a hard limit for how many values are allowed. Oracle’s limit is 1000.

In addition to the primitive values in the value list, you can create a java.util.List of object arrays. This list would support multiple expressions defined for the in clause such as select \* from T\_ACTOR where (id, last\_name) in ((1, 'Johnson'), (2, 'Harrop'\)). This of course requires that your database supports this syntax.

3.7.4. Handling complex types for stored procedure calls

When you call stored procedures you can sometimes use complex types specific to the database. To accommodate these types, Spring provides a SqlReturnType for handling them when they are returned from the stored procedure call and SqlTypeValue when they are passed in as a parameter to the stored procedure.

Here is an example of returning the value of an Oracle STRUCT object of the user declared type ITEM\_TYPE. The SqlReturnType interface has a single method named getTypeValue that must be implemented. This interface is used as part of the declaration of an SqlOutParameter.

public class TestItemStoredProcedure extends StoredProcedure {

public TestItemStoredProcedure(DataSource dataSource) {

...

declareParameter(new SqlOutParameter("item", OracleTypes.STRUCT, "ITEM\_TYPE",

new SqlReturnType() {

public Object getTypeValue(CallableStatement cs, int colIndx, int sqlType, String typeName) throws SQLException {

STRUCT struct = (STRUCT) cs.getObject(colIndx);

Object[] attr = struct.getAttributes();

TestItem item = new TestItem();

item.setId(((Number) attr[0]).longValue());

item.setDescription((String) attr[1]);

item.setExpirationDate((java.util.Date) attr[2]);

return item;

}

}));

...

}

You use the SqlTypeValue to pass in the value of a Java object like TestItem into a stored procedure. The SqlTypeValue interface has a single method named createTypeValue that you must implement. The active connection is passed in, and you can use it to create database-specific objects such as StructDescriptors, as shown in the following example, or ArrayDescriptors.

final TestItem testItem = new TestItem(123L, "A test item",

new SimpleDateFormat("yyyy-M-d").parse("2010-12-31"));

SqlTypeValue value = new AbstractSqlTypeValue() {

protected Object createTypeValue(Connection conn, int sqlType, String typeName) throws SQLException {

StructDescriptor itemDescriptor = new StructDescriptor(typeName, conn);

Struct item = new STRUCT(itemDescriptor, conn,

new Object[] {

testItem.getId(),

testItem.getDescription(),

new java.sql.Date(testItem.getExpirationDate().getTime())

});

return item;

}

};

This SqlTypeValue can now be added to the Map containing the input parameters for the execute call of the stored procedure.

Another use for the SqlTypeValue is passing in an array of values to an Oracle stored procedure. Oracle has its own internal ARRAY class that must be used in this case, and you can use the SqlTypeValue to create an instance of the Oracle ARRAY and populate it with values from the Java ARRAY.

final Long[] ids = new Long[] {1L, 2L};

SqlTypeValue value = new AbstractSqlTypeValue() {

protected Object createTypeValue(Connection conn, int sqlType, String typeName) throws SQLException {

ArrayDescriptor arrayDescriptor = new ArrayDescriptor(typeName, conn);

ARRAY idArray = new ARRAY(arrayDescriptor, conn, ids);

return idArray;

}

};

3.8. Embedded database support

The org.springframework.jdbc.datasource.embedded package provides support for embedded Java database engines. Support for HSQL, H2, and Derby is provided natively. You can also use an extensible API to plug in new embedded database types and DataSource implementations.

3.8.1. Why use an embedded database?

An embedded database is useful during the development phase of a project because of its lightweight nature. Benefits include ease of configuration, quick startup time, testability, and the ability to rapidly evolve SQL during development.

3.8.2. Creating an embedded database using Spring XML

If you want to expose an embedded database instance as a bean in a Spring ApplicationContext, use the embedded-database tag in the spring-jdbc namespace:

<jdbc:embedded-database id="dataSource" generate-name="true">

<jdbc:script location="classpath:schema.sql"/>

<jdbc:script location="classpath:test-data.sql"/>

</jdbc:embedded-database>

The preceding configuration creates an embedded HSQL database populated with SQL from schema.sql and test-data.sql resources in the root of the classpath. In addition, as a best practice, the embedded database will be assigned a uniquely generated name. The embedded database is made available to the Spring container as a bean of type javax.sql.DataSource which can then be injected into data access objects as needed.

3.8.3. Creating an embedded database programmatically

The EmbeddedDatabaseBuilder class provides a fluent API for constructing an embedded database programmatically. Use this when you need to create an embedded database in a standalone environment or in a standalone integration test like in the following example.

EmbeddedDatabase db = new EmbeddedDatabaseBuilder()

.generateUniqueName(true)

.setType(H2)

.setScriptEncoding("UTF-8")

.ignoreFailedDrops(true)

.addScript("schema.sql")

.addScripts("user\_data.sql", "country\_data.sql")

.build();

// perform actions against the db (EmbeddedDatabase extends javax.sql.DataSource)

db.shutdown()

Consult the Javadoc for EmbeddedDatabaseBuilder for further details on all supported options.

The EmbeddedDatabaseBuilder can also be used to create an embedded database using Java Config like in the following example.

@Configuration

public class DataSourceConfig {

@Bean

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder()

.generateUniqueName(true)

.setType(H2)

.setScriptEncoding("UTF-8")

.ignoreFailedDrops(true)

.addScript("schema.sql")

.addScripts("user\_data.sql", "country\_data.sql")

.build();

}

}

3.8.4. Selecting the embedded database type

Using HSQL

Spring supports HSQL 1.8.0 and above. HSQL is the default embedded database if no type is specified explicitly. To specify HSQL explicitly, set the type attribute of the embedded-database tag to HSQL. If you are using the builder API, call the setType(EmbeddedDatabaseType) method with EmbeddedDatabaseType.HSQL.

Using H2

Spring supports the H2 database as well. To enable H2, set the type attribute of the embedded-database tag to H2. If you are using the builder API, call the setType(EmbeddedDatabaseType) method with EmbeddedDatabaseType.H2.

Using Derby

Spring also supports Apache Derby 10.5 and above. To enable Derby, set the type attribute of the embedded-database tag to DERBY. If you are using the builder API, call the setType(EmbeddedDatabaseType) method with EmbeddedDatabaseType.DERBY.

3.8.5. Testing data access logic with an embedded database

Embedded databases provide a lightweight way to test data access code. The following is a data access integration test template that uses an embedded database. Using a template like this can be useful for one-offs when the embedded database does not need to be reused across test classes. However, if you wish to create an embedded database that is shared within a test suite, consider using the Spring TestContext Framework and configuring the embedded database as a bean in the Spring ApplicationContext as described in Creating an embedded database using Spring XML and Creating an embedded database programmatically.

public class DataAccessIntegrationTestTemplate {

private EmbeddedDatabase db;

@Before

public void setUp() {

// creates an HSQL in-memory database populated from default scripts

// classpath:schema.sql and classpath:data.sql

db = new EmbeddedDatabaseBuilder()

.generateUniqueName(true)

.addDefaultScripts()

.build();

}

@Test

public void testDataAccess() {

JdbcTemplate template = new JdbcTemplate(db);

template.query( /\* ... \*/ );

}

@After

public void tearDown() {

db.shutdown();

}

}

3.8.6. Generating unique names for embedded databases

Development teams often encounter errors with embedded databases if their test suite inadvertently attempts to recreate additional instances of the same database. This can happen quite easily if an XML configuration file or @Configuration class is responsible for creating an embedded database and the corresponding configuration is then reused across multiple testing scenarios within the same test suite (i.e., within the same JVM process) –- for example, integration tests against embedded databases whose ApplicationContext configuration only differs with regard to which bean definition profiles are active.

The root cause of such errors is the fact that Spring’s EmbeddedDatabaseFactory (used internally by both the <jdbc:embedded-database> XML namespace element and the EmbeddedDatabaseBuilder for Java Config) will set the name of the embedded database to "testdb" if not otherwise specified. For the case of <jdbc:embedded-database>, the embedded database is typically assigned a name equal to the bean’s id (i.e., often something like "dataSource"). Thus, subsequent attempts to create an embedded database will not result in a new database. Instead, the same JDBC connection URL will be reused, and attempts to create a new embedded database will actually point to an existing embedded database created from the same configuration.

To address this common issue Spring Framework 4.2 provides support for generating unique names for embedded databases. To enable the use of generated names, use one of the following options.

EmbeddedDatabaseFactory.setGenerateUniqueDatabaseName()

EmbeddedDatabaseBuilder.generateUniqueName()

<jdbc:embedded-database generate-name="true" …​ >

3.8.7. Extending the embedded database support

Spring JDBC embedded database support can be extended in two ways:

Implement EmbeddedDatabaseConfigurer to support a new embedded database type.

Implement DataSourceFactory to support a new DataSource implementation, such as a connection pool to manage embedded database connections.

You are encouraged to contribute back extensions to the Spring community at jira.spring.io.

3.9. Initializing a DataSource

The org.springframework.jdbc.datasource.init package provides support for initializing an existing DataSource. The embedded database support provides one option for creating and initializing a DataSource for an application, but sometimes you need to initialize an instance running on a server somewhere.

3.9.1. Initializing a database using Spring XML

If you want to initialize a database and you can provide a reference to a DataSource bean, use the initialize-database tag in the spring-jdbc namespace:

<jdbc:initialize-database data-source="dataSource">

<jdbc:script location="classpath:com/foo/sql/db-schema.sql"/>

<jdbc:script location="classpath:com/foo/sql/db-test-data.sql"/>

</jdbc:initialize-database>

The example above executes the two scripts specified against the database: the first script creates a schema, and the second populates tables with a test data set. The script locations can also be patterns with wildcards in the usual ant style used for resources in Spring (e.g. classpath\*:/com/foo/\*\*/sql/\*-data.sql). If a pattern is used, the scripts are executed in lexical order of their URL or filename.

The default behavior of the database initializer is to unconditionally execute the scripts provided. This will not always be what you want, for instance, if you are executing the scripts against a database that already has test data in it. The likelihood of accidentally deleting data is reduced by following the common pattern (as shown above) of creating the tables first and then inserting the data — the first step will fail if the tables already exist.

However, to gain more control over the creation and deletion of existing data, the XML namespace provides a few additional options. The first is a flag to switch the initialization on and off. This can be set according to the environment (e.g. to pull a boolean value from system properties or an environment bean), for example:

<jdbc:initialize-database data-source="dataSource"

enabled="#{systemProperties.INITIALIZE\_DATABASE}">

<jdbc:script location="..."/>

</jdbc:initialize-database>

The second option to control what happens with existing data is to be more tolerant of failures. To this end you can control the ability of the initializer to ignore certain errors in the SQL it executes from the scripts, for example:

<jdbc:initialize-database data-source="dataSource" ignore-failures="DROPS">

<jdbc:script location="..."/>

</jdbc:initialize-database>

In this example we are saying we expect that sometimes the scripts will be executed against an empty database, and there are some DROP statements in the scripts which would therefore fail. So failed SQL DROP statements will be ignored, but other failures will cause an exception. This is useful if your SQL dialect doesn’t support DROP …​ IF EXISTS (or similar) but you want to unconditionally remove all test data before re-creating it. In that case the first script is usually a set of DROP statements, followed by a set of CREATE statements.

The ignore-failures option can be set to NONE (the default), DROPS (ignore failed drops), or ALL (ignore all failures).

Each statement should be separated by ; or a new line if the ; character is not present at all in the script. You can control that globally or script by script, for example:

<jdbc:initialize-database data-source="dataSource" separator="@@">

<jdbc:script location="classpath:com/foo/sql/db-schema.sql" separator=";"/>

<jdbc:script location="classpath:com/foo/sql/db-test-data-1.sql"/>

<jdbc:script location="classpath:com/foo/sql/db-test-data-2.sql"/>

</jdbc:initialize-database>

In this example, the two test-data scripts use @@ as statement separator and only the db-schema.sql uses ;. This configuration specifies that the default separator is @@ and override that default for the db-schema script.

If you need more control than you get from the XML namespace, you can simply use the DataSourceInitializer directly and define it as a component in your application.

Initialization of other components that depend on the database

A large class of applications can just use the database initializer with no further complications: those that do not use the database until after the Spring context has started. If your application is not one of those then you might need to read the rest of this section.

The database initializer depends on a DataSource instance and executes the scripts provided in its initialization callback (analogous to an init-method in an XML bean definition, a @PostConstruct method in a component, or the afterPropertiesSet() method in a component that implements InitializingBean). If other beans depend on the same data source and also use the data source in an initialization callback, then there might be a problem because the data has not yet been initialized. A common example of this is a cache that initializes eagerly and loads data from the database on application startup.

To get around this issue you have two options: change your cache initialization strategy to a later phase, or ensure that the database initializer is initialized first.

The first option might be easy if the application is in your control, and not otherwise. Some suggestions for how to implement this include:

Make the cache initialize lazily on first usage, which improves application startup time.

Have your cache or a separate component that initializes the cache implement Lifecycle or SmartLifecycle. When the application context starts up a SmartLifecycle can be automatically started if its autoStartup flag is set, and a Lifecycle can be started manually by calling ConfigurableApplicationContext.start() on the enclosing context.

Use a Spring ApplicationEvent or similar custom observer mechanism to trigger the cache initialization. ContextRefreshedEvent is always published by the context when it is ready for use (after all beans have been initialized), so that is often a useful hook (this is how the SmartLifecycle works by default).

The second option can also be easy. Some suggestions on how to implement this include:

Rely on the default behavior of the Spring BeanFactory, which is that beans are initialized in registration order. You can easily arrange that by adopting the common practice of a set of <import/> elements in XML configuration that order your application modules, and ensure that the database and database initialization are listed first.

Separate the DataSource and the business components that use it, and control their startup order by putting them in separate ApplicationContext instances (e.g. the parent context contains the DataSource, and child context contains the business components). This structure is common in Spring web applications but can be more generally applied.

4. Object Relational Mapping (ORM) Data Access

4.1. Introduction to ORM with Spring

The Spring Framework supports integration with the Java Persistence API (JPA) as well as native Hibernate for resource management, data access object (DAO) implementations, and transaction strategies. For example, for Hibernate there is first-class support with several convenient IoC features that address many typical Hibernate integration issues. You can configure all of the supported features for O/R (object relational) mapping tools through Dependency Injection. They can participate in Spring’s resource and transaction management, and they comply with Spring’s generic transaction and DAO exception hierarchies. The recommended integration style is to code DAOs against plain Hibernate or JPA APIs.

Spring adds significant enhancements to the ORM layer of your choice when you create data access applications. You can leverage as much of the integration support as you wish, and you should compare this integration effort with the cost and risk of building a similar infrastructure in-house. You can use much of the ORM support as you would a library, regardless of technology, because everything is designed as a set of reusable JavaBeans. ORM in a Spring IoC container facilitates configuration and deployment. Thus most examples in this section show configuration inside a Spring container.

Benefits of using the Spring Framework to create your ORM DAOs include:

Easier testing. Spring’s IoC approach makes it easy to swap the implementations and configuration locations of Hibernate SessionFactory instances, JDBC DataSource instances, transaction managers, and mapped object implementations (if needed). This in turn makes it much easier to test each piece of persistence-related code in isolation.

Common data access exceptions. Spring can wrap exceptions from your ORM tool, converting them from proprietary (potentially checked) exceptions to a common runtime DataAccessException hierarchy. This feature allows you to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches, throws, and exception declarations. You can still trap and handle exceptions as necessary. Remember that JDBC exceptions (including DB-specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.

General resource management. Spring application contexts can handle the location and configuration of Hibernate SessionFactory instances, JPA EntityManagerFactory instances, JDBC DataSource instances, and other related resources. This makes these values easy to manage and change. Spring offers efficient, easy, and safe handling of persistence resources. For example, related code that uses Hibernate generally needs to use the same Hibernate Session to ensure efficiency and proper transaction handling. Spring makes it easy to create and bind a Session to the current thread transparently, by exposing a current Session through the Hibernate SessionFactory. Thus Spring solves many chronic problems of typical Hibernate usage, for any local or JTA transaction environment.

Integrated transaction management. You can wrap your ORM code with a declarative, aspect-oriented programming (AOP) style method interceptor either through the @Transactional annotation or by explicitly configuring the transaction AOP advice in an XML configuration file. In both cases, transaction semantics and exception handling (rollback, and so on) are handled for you. As discussed below, in Resource and transaction management, you can also swap various transaction managers, without affecting your ORM-related code. For example, you can swap between local transactions and JTA, with the same full services (such as declarative transactions) available in both scenarios. Additionally, JDBC-related code can fully integrate transactionally with the code you use to do ORM. This is useful for data access that is not suitable for ORM, such as batch processing and BLOB streaming, which still need to share common transactions with ORM operations.

For more comprehensive ORM support, including support for alternative database technologies such as MongoDB, you might want to check out the Spring Data suite of projects. If you are a JPA user, the Getting Started Accessing Data with JPA guide from https://spring.io provides a great introduction.

4.2. General ORM integration considerations

This section highlights considerations that apply to all ORM technologies. The Hibernate section provides more details and also show these features and configurations in a concrete context.

The major goal of Spring’s ORM integration is clear application layering, with any data access and transaction technology, and for loose coupling of application objects. No more business service dependencies on the data access or transaction strategy, no more hard-coded resource lookups, no more hard-to-replace singletons, no more custom service registries. One simple and consistent approach to wiring up application objects, keeping them as reusable and free from container dependencies as possible. All the individual data access features are usable on their own but integrate nicely with Spring’s application context concept, providing XML-based configuration and cross-referencing of plain JavaBean instances that need not be Spring-aware. In a typical Spring application, many important objects are JavaBeans: data access templates, data access objects, transaction managers, business services that use the data access objects and transaction managers, web view resolvers, web controllers that use the business services,and so on.

4.2.1. Resource and transaction management

Typical business applications are cluttered with repetitive resource management code. Many projects try to invent their own solutions, sometimes sacrificing proper handling of failures for programming convenience. Spring advocates simple solutions for proper resource handling, namely IoC through templating in the case of JDBC and applying AOP interceptors for the ORM technologies.

The infrastructure provides proper resource handling and appropriate conversion of specific API exceptions to an unchecked infrastructure exception hierarchy. Spring introduces a DAO exception hierarchy, applicable to any data access strategy. For direct JDBC, the JdbcTemplate class mentioned in a previous section provides connection handling and proper conversion of SQLException to the DataAccessException hierarchy, including translation of database-specific SQL error codes to meaningful exception classes. For ORM technologies, see the next section for how to get the same exception translation benefits.

When it comes to transaction management, the JdbcTemplate class hooks in to the Spring transaction support and supports both JTA and JDBC transactions, through respective Spring transaction managers. For the supported ORM technologies Spring offers Hibernate and JPA support through the Hibernate and JPA transaction managers as well as JTA support. For details on transaction support, see the Transaction Management chapter.

4.2.2. Exception translation

When you use Hibernate or JPA in a DAO, you must decide how to handle the persistence technology’s native exception classes. The DAO throws a subclass of a HibernateException or PersistenceException depending on the technology. These exceptions are all runtime exceptions and do not have to be declared or caught. You may also have to deal with IllegalArgumentException and IllegalStateException. This means that callers can only treat exceptions as generally fatal, unless they want to depend on the persistence technology’s own exception structure. Catching specific causes such as an optimistic locking failure is not possible without tying the caller to the implementation strategy. This trade-off might be acceptable to applications that are strongly ORM-based and/or do not need any special exception treatment. However, Spring enables exception translation to be applied transparently through the @Repository annotation:

@Repository

public class ProductDaoImpl implements ProductDao {

// class body here...

}

<beans>

<!-- Exception translation bean post processor -->

<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

<bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>

The postprocessor automatically looks for all exception translators (implementations of the PersistenceExceptionTranslator interface) and advises all beans marked with the @Repository annotation so that the discovered translators can intercept and apply the appropriate translation on the thrown exceptions.

In summary: you can implement DAOs based on the plain persistence technology’s API and annotations, while still benefiting from Spring-managed transactions, dependency injection, and transparent exception conversion (if desired) to Spring’s custom exception hierarchies.

4.3. Hibernate

We will start with a coverage of Hibernate 5 in a Spring environment, using it to demonstrate the approach that Spring takes towards integrating O/R mappers. This section will cover many issues in detail and show different variations of DAO implementations and transaction demarcation. Most of these patterns can be directly translated to all other supported ORM tools. The following sections in this chapter will then cover the other ORM technologies, showing briefer examples there.

As of Spring Framework 5.0, Spring requires Hibernate ORM 4.3 or later for JPA support and even Hibernate ORM 5.0+ for programming against the native Hibernate Session API. Note that the Hibernate team does not maintain any versions prior to 5.0 anymore and is likely to focus on 5.2+ exclusively soon.

4.3.1. SessionFactory setup in a Spring container

To avoid tying application objects to hard-coded resource lookups, you can define resources such as a JDBC DataSource or a Hibernate SessionFactory as beans in the Spring container. Application objects that need to access resources receive references to such predefined instances through bean references, as illustrated in the DAO definition in the next section.

The following excerpt from an XML application context definition shows how to set up a JDBC DataSource and a Hibernate SessionFactory on top of it:

<beans>

<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="org.hsqldb.jdbcDriver"/>

<property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>

<property name="username" value="sa"/>

<property name="password" value=""/>

</bean>

<bean id="mySessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

<property name="dataSource" ref="myDataSource"/>

<property name="mappingResources">

<list>

<value>product.hbm.xml</value>

</list>

</property>

<property name="hibernateProperties">

<value>

hibernate.dialect=org.hibernate.dialect.HSQLDialect

</value>

</property>

</bean>

</beans>

Switching from a local Jakarta Commons DBCP BasicDataSource to a JNDI-located DataSource (usually managed by an application server) is just a matter of configuration:

<beans>

<jee:jndi-lookup id="myDataSource" jndi-name="java:comp/env/jdbc/myds"/>

</beans>

You can also access a JNDI-located SessionFactory, using Spring’s JndiObjectFactoryBean / <jee:jndi-lookup> to retrieve and expose it. However, that is typically not common outside of an EJB context.

4.3.2. Implementing DAOs based on plain Hibernate API

Hibernate has a feature called contextual sessions, wherein Hibernate itself manages one current Session per transaction. This is roughly equivalent to Spring’s synchronization of one Hibernate Session per transaction. A corresponding DAO implementation resembles the following example, based on the plain Hibernate API:

public class ProductDaoImpl implements ProductDao {

private SessionFactory sessionFactory;

public void setSessionFactory(SessionFactory sessionFactory) {

this.sessionFactory = sessionFactory;

}

public Collection loadProductsByCategory(String category) {

return this.sessionFactory.getCurrentSession()

.createQuery("from test.Product product where product.category=?")

.setParameter(0, category)

.list();

}

}

This style is similar to that of the Hibernate reference documentation and examples, except for holding the SessionFactory in an instance variable. We strongly recommend such an instance-based setup over the old-school static HibernateUtil class from Hibernate’s CaveatEmptor sample application. (In general, do not keep any resources in static variables unless absolutely necessary.)

The above DAO follows the dependency injection pattern: it fits nicely into a Spring IoC container, just as it would if coded against Spring’s HibernateTemplate. Of course, such a DAO can also be set up in plain Java (for example, in unit tests). Simply instantiate it and call setSessionFactory(..) with the desired factory reference. As a Spring bean definition, the DAO would resemble the following:

<beans>

<bean id="myProductDao" class="product.ProductDaoImpl">

<property name="sessionFactory" ref="mySessionFactory"/>

</bean>

</beans>

The main advantage of this DAO style is that it depends on Hibernate API only; no import of any Spring class is required. This is of course appealing from a non-invasiveness perspective, and will no doubt feel more natural to Hibernate developers.

However, the DAO throws plain HibernateException (which is unchecked, so does not have to be declared or caught), which means that callers can only treat exceptions as generally fatal - unless they want to depend on Hibernate’s own exception hierarchy. Catching specific causes such as an optimistic locking failure is not possible without tying the caller to the implementation strategy. This trade off might be acceptable to applications that are strongly Hibernate-based and/or do not need any special exception treatment.

Fortunately, Spring’s LocalSessionFactoryBean supports Hibernate’s SessionFactory.getCurrentSession() method for any Spring transaction strategy, returning the current Spring-managed transactional Session even with HibernateTransactionManager. Of course, the standard behavior of that method remains the return of the current Session associated with the ongoing JTA transaction, if any. This behavior applies regardless of whether you are using Spring’s JtaTransactionManager, EJB container managed transactions (CMTs), or JTA.

In summary: you can implement DAOs based on the plain Hibernate API, while still being able to participate in Spring-managed transactions.

4.3.3. Declarative transaction demarcation

We recommend that you use Spring’s declarative transaction support, which enables you to replace explicit transaction demarcation API calls in your Java code with an AOP transaction interceptor. This transaction interceptor can be configured in a Spring container using either Java annotations or XML. This declarative transaction capability allows you to keep business services free of repetitive transaction demarcation code and to focus on adding business logic, which is the real value of your application.

Prior to continuing, you are strongly encouraged to read Declarative transaction management if you have not done so.

You may annotate the service layer with @Transactional annotations and instruct the Spring container to find these annotations and provide transactional semantics for these annotated methods.

public class ProductServiceImpl implements ProductService {

private ProductDao productDao;

public void setProductDao(ProductDao productDao) {

this.productDao = productDao;

}

@Transactional

public void increasePriceOfAllProductsInCategory(final String category) {

List productsToChange = this.productDao.loadProductsByCategory(category);

// ...

}

@Transactional(readOnly = true)

public List<Product> findAllProducts() {

return this.productDao.findAllProducts();

}

}

All you need to set up in the container is the PlatformTransactionManager implementation as a bean as well as a "<tx:annotation-driven/>" entry, opting into @Transactional processing at runtime.

<?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:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop

http://www.springframework.org/schema/aop/spring-aop.xsd">

<!-- SessionFactory, DataSource, etc. omitted -->

<bean id="transactionManager"

class="org.springframework.orm.hibernate5.HibernateTransactionManager">

<property name="sessionFactory" ref="sessionFactory"/>

</bean>

<tx:annotation-driven/>

<bean id="myProductService" class="product.SimpleProductService">

<property name="productDao" ref="myProductDao"/>

</bean>

</beans>

4.3.4. Programmatic transaction demarcation

You can demarcate transactions in a higher level of the application, on top of such lower-level data access services spanning any number of operations. Nor do restrictions exist on the implementation of the surrounding business service; it just needs a Spring PlatformTransactionManager. Again, the latter can come from anywhere, but preferably as a bean reference through a setTransactionManager(..) method, just as the productDAO should be set by a setProductDao(..) method. The following snippets show a transaction manager and a business service definition in a Spring application context, and an example for a business method implementation:

<beans>

<bean id="myTxManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">

<property name="sessionFactory" ref="mySessionFactory"/>

</bean>

<bean id="myProductService" class="product.ProductServiceImpl">

<property name="transactionManager" ref="myTxManager"/>

<property name="productDao" ref="myProductDao"/>

</bean>

</beans>

public class ProductServiceImpl implements ProductService {

private TransactionTemplate transactionTemplate;

private ProductDao productDao;

public void setTransactionManager(PlatformTransactionManager transactionManager) {

this.transactionTemplate = new TransactionTemplate(transactionManager);

}

public void setProductDao(ProductDao productDao) {

this.productDao = productDao;

}

public void increasePriceOfAllProductsInCategory(final String category) {

this.transactionTemplate.execute(new TransactionCallbackWithoutResult() {

public void doInTransactionWithoutResult(TransactionStatus status) {

List productsToChange = this.productDao.loadProductsByCategory(category);

// do the price increase...

}

});

}

}

Spring’s TransactionInterceptor allows any checked application exception to be thrown with the callback code, while TransactionTemplate is restricted to unchecked exceptions within the callback. TransactionTemplate triggers a rollback in case of an unchecked application exception, or if the transaction is marked rollback-only by the application (via TransactionStatus). TransactionInterceptor behaves the same way by default but allows configurable rollback policies per method.

4.3.5. Transaction management strategies

Both TransactionTemplate and TransactionInterceptor delegate the actual transaction handling to a PlatformTransactionManager instance, which can be a HibernateTransactionManager (for a single Hibernate SessionFactory, using a ThreadLocal Session under the hood) or a JtaTransactionManager (delegating to the JTA subsystem of the container) for Hibernate applications. You can even use a custom PlatformTransactionManager implementation. Switching from native Hibernate transaction management to JTA, such as when facing distributed transaction requirements for certain deployments of your application, is just a matter of configuration. Simply replace the Hibernate transaction manager with Spring’s JTA transaction implementation. Both transaction demarcation and data access code will work without changes, because they just use the generic transaction management APIs.

For distributed transactions across multiple Hibernate session factories, simply combine JtaTransactionManager as a transaction strategy with multiple LocalSessionFactoryBean definitions. Each DAO then gets one specific SessionFactory reference passed into its corresponding bean property. If all underlying JDBC data sources are transactional container ones, a business service can demarcate transactions across any number of DAOs and any number of session factories without special regard, as long as it is using JtaTransactionManager as the strategy.

Both HibernateTransactionManager and JtaTransactionManager allow for proper JVM-level cache handling with Hibernate, without container-specific transaction manager lookup or a JCA connector (if you are not using EJB to initiate transactions).

HibernateTransactionManager can export the Hibernate JDBC Connection to plain JDBC access code, for a specific DataSource. This capability allows for high-level transaction demarcation with mixed Hibernate and JDBC data access completely without JTA, if you are accessing only one database. HibernateTransactionManager automatically exposes the Hibernate transaction as a JDBC transaction if you have set up the passed-in SessionFactory with a DataSource through the dataSource property of the LocalSessionFactoryBean class. Alternatively, you can specify explicitly the DataSource for which the transactions are supposed to be exposed through the dataSource property of the HibernateTransactionManager class.

4.3.6. Comparing container-managed and locally defined resources

You can switch between a container-managed JNDI SessionFactory and a locally defined one, without having to change a single line of application code. Whether to keep resource definitions in the container or locally within the application is mainly a matter of the transaction strategy that you use. Compared to a Spring-defined local SessionFactory, a manually registered JNDI SessionFactory does not provide any benefits. Deploying a SessionFactory through Hibernate’s JCA connector provides the added value of participating in the Java EE server’s management infrastructure, but does not add actual value beyond that.

Spring’s transaction support is not bound to a container. Configured with any strategy other than JTA, transaction support also works in a stand-alone or test environment. Especially in the typical case of single-database transactions, Spring’s single-resource local transaction support is a lightweight and powerful alternative to JTA. When you use local EJB stateless session beans to drive transactions, you depend both on an EJB container and JTA, even if you access only a single database, and only use stateless session beans to provide declarative transactions through container-managed transactions. Also, direct use of JTA programmatically requires a Java EE environment as well. JTA does not involve only container dependencies in terms of JTA itself and of JNDI DataSource instances. For non-Spring, JTA-driven Hibernate transactions, you have to use the Hibernate JCA connector, or extra Hibernate transaction code with the TransactionManagerLookup configured for proper JVM-level caching.

Spring-driven transactions can work as well with a locally defined Hibernate SessionFactory as they do with a local JDBC DataSource if they are accessing a single database. Thus you only have to use Spring’s JTA transaction strategy when you have distributed transaction requirements. A JCA connector requires container-specific deployment steps, and obviously JCA support in the first place. This configuration requires more work than deploying a simple web application with local resource definitions and Spring-driven transactions. Also, you often need the Enterprise Edition of your container if you are using, for example, WebLogic Express, which does not provide JCA. A Spring application with local resources and transactions spanning one single database works in any Java EE web container (without JTA, JCA, or EJB) such as Tomcat, Resin, or even plain Jetty. Additionally, you can easily reuse such a middle tier in desktop applications or test suites.

All things considered, if you do not use EJBs, stick with local SessionFactory setup and Spring’s HibernateTransactionManager or JtaTransactionManager. You get all of the benefits, including proper transactional JVM-level caching and distributed transactions, without the inconvenience of container deployment. JNDI registration of a Hibernate SessionFactory through the JCA connector only adds value when used in conjunction with EJBs.

4.3.7. Spurious application server warnings with Hibernate

In some JTA environments with very strict XADataSource implementations — currently only some WebLogic Server and WebSphere versions — when Hibernate is configured without regard to the JTA PlatformTransactionManager object for that environment, it is possible for spurious warning or exceptions to show up in the application server log. These warnings or exceptions indicate that the connection being accessed is no longer valid, or JDBC access is no longer valid, possibly because the transaction is no longer active. As an example, here is an actual exception from WebLogic:

java.sql.SQLException: The transaction is no longer active - status: 'Committed'. No

further JDBC access is allowed within this transaction.

You resolve this warning by simply making Hibernate aware of the JTA PlatformTransactionManager instance, to which it will synchronize (along with Spring). You have two options for doing this:

If in your application context you are already directly obtaining the JTA PlatformTransactionManager object (presumably from JNDI through JndiObjectFactoryBean or <jee:jndi-lookup>) and feeding it, for example, to Spring’s JtaTransactionManager, then the easiest way is to specify a reference to the bean defining this JTA PlatformTransactionManager instance as the value of the jtaTransactionManager property for LocalSessionFactoryBean. Spring then makes the object available to Hibernate.

More likely you do not already have the JTA PlatformTransactionManager instance, because Spring’s JtaTransactionManager can find it itself. Thus you need to configure Hibernate to look up JTA PlatformTransactionManager directly. You do this by configuring an application server- specific TransactionManagerLookup class in the Hibernate configuration, as described in the Hibernate manual.

The remainder of this section describes the sequence of events that occur with and without Hibernate’s awareness of the JTA PlatformTransactionManager.

When Hibernate is not configured with any awareness of the JTA PlatformTransactionManager, the following events occur when a JTA transaction commits:

The JTA transaction commits.

Spring’s JtaTransactionManager is synchronized to the JTA transaction, so it is called back through an afterCompletion callback by the JTA transaction manager.

Among other activities, this synchronization can trigger a callback by Spring to Hibernate, through Hibernate’s afterTransactionCompletion callback (used to clear the Hibernate cache), followed by an explicit close() call on the Hibernate Session, which causes Hibernate to attempt to close() the JDBC Connection.

In some environments, this Connection.close() call then triggers the warning or error, as the application server no longer considers the Connection usable at all, because the transaction has already been committed.

When Hibernate is configured with awareness of the JTA PlatformTransactionManager, the following events occur when a JTA transaction commits:

the JTA transaction is ready to commit.

Spring’s JtaTransactionManager is synchronized to the JTA transaction, so the transaction is called back through a beforeCompletion callback by the JTA transaction manager.

Spring is aware that Hibernate itself is synchronized to the JTA transaction, and behaves differently than in the previous scenario. Assuming the Hibernate Session needs to be closed at all, Spring will close it now.

The JTA transaction commits.

Hibernate is synchronized to the JTA transaction, so the transaction is called back through an afterCompletion callback by the JTA transaction manager, and can properly clear its cache.

4.4. JPA

The Spring JPA, available under the org.springframework.orm.jpa package, offers comprehensive support for the Java Persistence API in a similar manner to the integration with Hibernate, while being aware of the underlying implementation in order to provide additional features.

4.4.1. Three options for JPA setup in a Spring environment

The Spring JPA support offers three ways of setting up the JPA EntityManagerFactory that will be used by the application to obtain an entity manager.

LocalEntityManagerFactoryBean

Only use this option in simple deployment environments such as stand-alone applications and integration tests.

The LocalEntityManagerFactoryBean creates an EntityManagerFactory suitable for simple deployment environments where the application uses only JPA for data access. The factory bean uses the JPA PersistenceProvider autodetection mechanism (according to JPA’s Java SE bootstrapping) and, in most cases, requires you to specify only the persistence unit name:

<beans>

<bean id="myEmf" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">

<property name="persistenceUnitName" value="myPersistenceUnit"/>

</bean>

</beans>

This form of JPA deployment is the simplest and the most limited. You cannot refer to an existing JDBC DataSource bean definition and no support for global transactions exists. Furthermore, weaving (byte-code transformation) of persistent classes is provider-specific, often requiring a specific JVM agent to specified on startup. This option is sufficient only for stand-alone applications and test environments, for which the JPA specification is designed.

Obtaining an EntityManagerFactory from JNDI

Use this option when deploying to a Java EE server. Check your server’s documentation on how to deploy a custom JPA provider into your server, allowing for a different provider than the server’s default.

Obtaining an EntityManagerFactory from JNDI (for example in a Java EE environment), is simply a matter of changing the XML configuration:

<beans>

<jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersistenceUnit"/>

</beans>

This action assumes standard Java EE bootstrapping: the Java EE server autodetects persistence units (in effect, META-INF/persistence.xml files in application jars) and persistence-unit-ref entries in the Java EE deployment descriptor (for example, web.xml) and defines environment naming context locations for those persistence units.

In such a scenario, the entire persistence unit deployment, including the weaving (byte-code transformation) of persistent classes, is up to the Java EE server. The JDBC DataSource is defined through a JNDI location in the META-INF/persistence.xml file; EntityManager transactions are integrated with the server’s JTA subsystem. Spring merely uses the obtained EntityManagerFactory, passing it on to application objects through dependency injection, and managing transactions for the persistence unit, typically through JtaTransactionManager.

If multiple persistence units are used in the same application, the bean names of such JNDI-retrieved persistence units should match the persistence unit names that the application uses to refer to them, for example, in @PersistenceUnit and @PersistenceContext annotations.

LocalContainerEntityManagerFactoryBean

Use this option for full JPA capabilities in a Spring-based application environment. This includes web containers such as Tomcat as well as stand-alone applications and integration tests with sophisticated persistence requirements.

The LocalContainerEntityManagerFactoryBean gives full control over EntityManagerFactory configuration and is appropriate for environments where fine-grained customization is required. The LocalContainerEntityManagerFactoryBean creates a PersistenceUnitInfo instance based on the persistence.xml file, the supplied dataSourceLookup strategy, and the specified loadTimeWeaver. It is thus possible to work with custom data sources outside of JNDI and to control the weaving process. The following example shows a typical bean definition for a LocalContainerEntityManagerFactoryBean:

<beans>

<bean id="myEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

<property name="dataSource" ref="someDataSource"/>

<property name="loadTimeWeaver">

<bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>

</property>

</bean>

</beans>

The following example shows a typical persistence.xml file:

<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">

<persistence-unit name="myUnit" transaction-type="RESOURCE\_LOCAL">

<mapping-file>META-INF/orm.xml</mapping-file>

<exclude-unlisted-classes/>

</persistence-unit>

</persistence>

The <exclude-unlisted-classes/> shortcut indicates that no scanning for annotated entity classes is supposed to occur. An explicit 'true' value specified - <exclude-unlisted-classes>true</exclude-unlisted-classes/> - also means no scan. <exclude-unlisted-classes>false</exclude-unlisted-classes/> does trigger a scan; however, it is recommended to simply omit the exclude-unlisted-classes element if you want entity class scanning to occur.

Using the LocalContainerEntityManagerFactoryBean is the most powerful JPA setup option, allowing for flexible local configuration within the application. It supports links to an existing JDBC DataSource, supports both local and global transactions, and so on. However, it also imposes requirements on the runtime environment, such as the availability of a weaving-capable class loader if the persistence provider demands byte-code transformation.

This option may conflict with the built-in JPA capabilities of a Java EE server. In a full Java EE environment, consider obtaining your EntityManagerFactory from JNDI. Alternatively, specify a custom persistenceXmlLocation on your LocalContainerEntityManagerFactoryBean definition, for example, META-INF/my-persistence.xml, and only include a descriptor with that name in your application jar files. Because the Java EE server only looks for default META-INF/persistence.xml files, it ignores such custom persistence units and hence avoid conflicts with a Spring-driven JPA setup upfront. (This applies to Resin 3.1, for example.)

When is load-time weaving required?

Not all JPA providers require a JVM agent. Hibernate is an example of one that does not. If your provider does not require an agent or you have other alternatives, such as applying enhancements at build time through a custom compiler or an ant task, the load-time weaver should not be used.

The LoadTimeWeaver interface is a Spring-provided class that allows JPA ClassTransformer instances to be plugged in a specific manner, depending whether the environment is a web container or application server. Hooking ClassTransformers through an agent typically is not efficient. The agents work against the entire virtual machine and inspect every class that is loaded, which is usually undesirable in a production server environment.

Spring provides a number of LoadTimeWeaver implementations for various environments, allowing ClassTransformer instances to be applied only per class loader and not per VM.

Refer to Spring configuration in the AOP chapter for more insight regarding the LoadTimeWeaver implementations and their setup, either generic or customized to various platforms (such as Tomcat, WebLogic, GlassFish, Resin and JBoss).

As described in the aforementioned section, you can configure a context-wide LoadTimeWeaver using the @EnableLoadTimeWeaving annotation of context:load-time-weaver XML element. Such a global weaver is picked up by all JPA LocalContainerEntityManagerFactoryBeans automatically. This is the preferred way of setting up a load-time weaver, delivering autodetection of the platform (WebLogic, GlassFish, Tomcat, Resin, JBoss or VM agent) and automatic propagation of the weaver to all weaver-aware beans:

<context:load-time-weaver/>

<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

...

</bean>

However, if needed, one can manually specify a dedicated weaver through the loadTimeWeaver property:

<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

<property name="loadTimeWeaver">

<bean class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>

</property>

</bean>

No matter how the LTW is configured, using this technique, JPA applications relying on instrumentation can run in the target platform (ex: Tomcat) without needing an agent. This is important especially when the hosting applications rely on different JPA implementations because the JPA transformers are applied only at class loader level and thus are isolated from each other.

Dealing with multiple persistence units

For applications that rely on multiple persistence units locations, stored in various JARS in the classpath, for example, Spring offers the PersistenceUnitManager to act as a central repository and to avoid the persistence units discovery process, which can be expensive. The default implementation allows multiple locations to be specified that are parsed and later retrieved through the persistence unit name. (By default, the classpath is searched for META-INF/persistence.xml files.)

<bean id="pum" class="org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManager">

<property name="persistenceXmlLocations">

<list>

<value>org/springframework/orm/jpa/domain/persistence-multi.xml</value>

<value>classpath:/my/package/\*\*/custom-persistence.xml</value>

<value>classpath\*:META-INF/persistence.xml</value>

</list>

</property>

<property name="dataSources">

<map>

<entry key="localDataSource" value-ref="local-db"/>

<entry key="remoteDataSource" value-ref="remote-db"/>

</map>

</property>

<!-- if no datasource is specified, use this one -->

<property name="defaultDataSource" ref="remoteDataSource"/>

</bean>

<bean id="emf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

<property name="persistenceUnitManager" ref="pum"/>

<property name="persistenceUnitName" value="myCustomUnit"/>

</bean>

The default implementation allows customization of the PersistenceUnitInfo instances, before they are fed to the JPA provider, declaratively through its properties, which affect all hosted units, or programmatically, through the PersistenceUnitPostProcessor, which allows persistence unit selection. If no PersistenceUnitManager is specified, one is created and used internally by LocalContainerEntityManagerFactoryBean.

4.4.2. Implementing DAOs based on JPA: EntityManagerFactory and EntityManager

Although EntityManagerFactory instances are thread-safe, EntityManager instances are not. The injected JPA EntityManager behaves like an EntityManager fetched from an application server’s JNDI environment, as defined by the JPA specification. It delegates all calls to the current transactional EntityManager, if any; otherwise, it falls back to a newly created EntityManager per operation, in effect making its usage thread-safe.

It is possible to write code against the plain JPA without any Spring dependencies, by using an injected EntityManagerFactory or EntityManager. Spring can understand @PersistenceUnit and @PersistenceContext annotations both at field and method level if a PersistenceAnnotationBeanPostProcessor is enabled. A plain JPA DAO implementation using the @PersistenceUnit annotation might look like this:

public class ProductDaoImpl implements ProductDao {

private EntityManagerFactory emf;

@PersistenceUnit

public void setEntityManagerFactory(EntityManagerFactory emf) {

this.emf = emf;

}

public Collection loadProductsByCategory(String category) {

EntityManager em = this.emf.createEntityManager();

try {

Query query = em.createQuery("from Product as p where p.category = ?1");

query.setParameter(1, category);

return query.getResultList();

}

finally {

if (em != null) {

em.close();

}

}

}

}

The DAO above has no dependency on Spring and still fits nicely into a Spring application context. Moreover, the DAO takes advantage of annotations to require the injection of the default EntityManagerFactory:

<beans>

<!-- bean post-processor for JPA annotations -->

<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>

<bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>

As an alternative to defining a PersistenceAnnotationBeanPostProcessor explicitly, consider using the Spring context:annotation-config XML element in your application context configuration. Doing so automatically registers all Spring standard post-processors for annotation-based configuration, including CommonAnnotationBeanPostProcessor and so on.

<beans>

<!-- post-processors for all standard config annotations -->

<context:annotation-config/>

<bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>

The main problem with such a DAO is that it always creates a new EntityManager through the factory. You can avoid this by requesting a transactional EntityManager (also called "shared EntityManager" because it is a shared, thread-safe proxy for the actual transactional EntityManager) to be injected instead of the factory:

public class ProductDaoImpl implements ProductDao {

@PersistenceContext

private EntityManager em;

public Collection loadProductsByCategory(String category) {

Query query = em.createQuery("from Product as p where p.category = :category");

query.setParameter("category", category);

return query.getResultList();

}

}

The @PersistenceContext annotation has an optional attribute type, which defaults to PersistenceContextType.TRANSACTION. This default is what you need to receive a shared EntityManager proxy. The alternative, PersistenceContextType.EXTENDED, is a completely different affair: This results in a so-called extended EntityManager, which is not thread-safe and hence must not be used in a concurrently accessed component such as a Spring-managed singleton bean. Extended EntityManagers are only supposed to be used in stateful components that, for example, reside in a session, with the lifecycle of the EntityManager not tied to a current transaction but rather being completely up to the application.

Method- and field-level Injection

Annotations that indicate dependency injections (such as @PersistenceUnit and @PersistenceContext) can be applied on field or methods inside a class, hence the expressions method-level injection and field-level injection. Field-level annotations are concise and easier to use while method-level allows for further processing of the injected dependency. In both cases the member visibility (public, protected, private) does not matter.

What about class-level annotations?

On the Java EE platform, they are used for dependency declaration and not for resource injection.

The injected EntityManager is Spring-managed (aware of the ongoing transaction). It is important to note that even though the new DAO implementation uses method level injection of an EntityManager instead of an EntityManagerFactory, no change is required in the application context XML due to annotation usage.

The main advantage of this DAO style is that it only depends on Java Persistence API; no import of any Spring class is required. Moreover, as the JPA annotations are understood, the injections are applied automatically by the Spring container. This is appealing from a non-invasiveness perspective, and might feel more natural to JPA developers.

4.4.3. Spring-driven JPA transactions

You are strongly encouraged to read Declarative transaction management if you have not done so, to get a more detailed coverage of Spring’s declarative transaction support.

The recommended strategy for JPA is local transactions via JPA’s native transaction support. Spring’s JpaTransactionManager provides many capabilities known from local JDBC transactions, such as transaction-specific isolation levels and resource-level read-only optimizations, against any regular JDBC connection pool (no XA requirement).

Spring JPA also allows a configured JpaTransactionManager to expose a JPA transaction to JDBC access code that accesses the same DataSource, provided that the registered JpaDialect supports retrieval of the underlying JDBC Connection. Out of the box, Spring provides dialects for the EclipseLink and Hibernate JPA implementations. See the next section for details on the JpaDialect mechanism.

4.4.4. JpaDialect and JpaVendorAdapter

As an advanced feature JpaTransactionManager and subclasses of AbstractEntityManagerFactoryBean support a custom JpaDialect, to be passed into the jpaDialect bean property. A JpaDialect implementation can enable some advanced features supported by Spring, usually in a vendor-specific manner:

Applying specific transaction semantics such as custom isolation level or transaction timeout)

Retrieving the transactional JDBC Connection for exposure to JDBC-based DAOs)

Advanced translation of PersistenceExceptions to Spring DataAccessExceptions

This is particularly valuable for special transaction semantics and for advanced translation of exception. The default implementation used (DefaultJpaDialect) does not provide any special capabilities and if the above features are required, you have to specify the appropriate dialect.

As an even broader provider adaptation facility primarily for Spring’s full-featured LocalContainerEntityManagerFactoryBean setup, JpaVendorAdapter combines the capabilities of JpaDialect with other provider-specific defaults. Specifying a HibernateJpaVendorAdapter or EclipseLinkJpaVendorAdapter is the most convenient way of auto-configuring an EntityManagerFactory setup for Hibernate or EclipseLink, respectively. Note that those provider adapters are primarily designed for use with Spring-driven transaction management, i.e. for use with JpaTransactionManager.

See the JpaDialect and JpaVendorAdapter javadocs for more details of its operations and how they are used within Spring’s JPA support.

4.4.5. Setting up JPA with JTA transaction management

As an alternative to JpaTransactionManager, Spring also allows for multi-resource transaction coordination via JTA, either in a Java EE environment or with a standalone transaction coordinator such as Atomikos. Aside from choosing Spring’s JtaTransactionManager instead of JpaTransactionManager, there are a few further steps to take:

The underlying JDBC connection pools need to be XA-capable and integrated with your transaction coordinator. This is usually straightforward in a Java EE environment, simply exposing a different kind of DataSource via JNDI. Check your application server documentation for details. Analogously, a standalone transaction coordinator usually comes with special XA-integrated DataSource implementations; again, check its docs.

The JPA EntityManagerFactory setup needs to be configured for JTA. This is provider-specific, typically via special properties to be specified as "jpaProperties" on LocalContainerEntityManagerFactoryBean. In the case of Hibernate, these properties are even version-specific; please check your Hibernate documentation for details.

Spring’s HibernateJpaVendorAdapter enforces certain Spring-oriented defaults such as the connection release mode "on-close" which matches Hibernate’s own default in Hibernate 5.0 but not anymore in 5.1/5.2. For a JTA setup, either do not declare HibernateJpaVendorAdapter to begin with, or turn off its prepareConnection flag. Alternatively, set Hibernate 5.2’s "hibernate.connection.handling\_mode" property to "DELAYED\_ACQUISITION\_AND\_RELEASE\_AFTER\_STATEMENT" to restore Hibernate’s own default. See Spurious application server warnings with Hibernate for a related note about WebLogic.

Alternatively, consider obtaining the EntityManagerFactory from your application server itself, i.e. via a JNDI lookup instead of a locally declared LocalContainerEntityManagerFactoryBean. A server-provided EntityManagerFactory might require special definitions in your server configuration, making the deployment less portable, but will be set up for the server’s JTA environment out of the box.

5. Marshalling XML using O/X Mappers

5.1. Introduction

In this chapter, we will describe Spring’s Object/XML Mapping support. Object/XML Mapping, or O/X mapping for short, is the act of converting an XML document to and from an object. This conversion process is also known as XML Marshalling, or XML Serialization. This chapter uses these terms interchangeably.

Within the field of O/X mapping, a marshaller is responsible for serializing an object (graph) to XML. In similar fashion, an unmarshaller deserializes the XML to an object graph. This XML can take the form of a DOM document, an input or output stream, or a SAX handler.

Some of the benefits of using Spring for your O/X mapping needs are:

5.1.1. Ease of configuration

Spring’s bean factory makes it easy to configure marshallers, without needing to construct JAXB context, JiBX binding factories, etc. The marshallers can be configured as any other bean in your application context. Additionally, XML namespace-based configuration is available for a number of marshallers, making the configuration even simpler.

5.1.2. Consistent interfaces

Spring’s O/X mapping operates through two global interfaces: the Marshaller and Unmarshaller interface. These abstractions allow you to switch O/X mapping frameworks with relative ease, with little or no changes required on the classes that do the marshalling. This approach has the additional benefit of making it possible to do XML marshalling with a mix-and-match approach (e.g. some marshalling performed using JAXB, other using Castor) in a non-intrusive fashion, leveraging the strength of each technology.

5.1.3. Consistent exception hierarchy

Spring provides a conversion from exceptions from the underlying O/X mapping tool to its own exception hierarchy with the XmlMappingException as the root exception. As can be expected, these runtime exceptions wrap the original exception so no information is lost.

5.2. Marshaller and Unmarshaller

As stated in the introduction, a marshaller serializes an object to XML, and an unmarshaller deserializes XML stream to an object. In this section, we will describe the two Spring interfaces used for this purpose.

5.2.1. Marshaller

Spring abstracts all marshalling operations behind the org.springframework.oxm.Marshaller interface, the main method of which is shown below.

public interface Marshaller {

/\*\*

\* Marshal the object graph with the given root into the provided Result.

\*/

void marshal(Object graph, Result result) throws XmlMappingException, IOException;

}

The Marshaller interface has one main method, which marshals the given object to a given javax.xml.transform.Result. Result is a tagging interface that basically represents an XML output abstraction: concrete implementations wrap various XML representations, as indicated in the table below.

Result implementation Wraps XML representation

DOMResult

org.w3c.dom.Node

SAXResult

org.xml.sax.ContentHandler

StreamResult

java.io.File, java.io.OutputStream, or java.io.Writer

Although the marshal() method accepts a plain object as its first parameter, most Marshaller implementations cannot handle arbitrary objects. Instead, an object class must be mapped in a mapping file, marked with an annotation, registered with the marshaller, or have a common base class. Refer to the further sections in this chapter to determine how your O/X technology of choice manages this.

5.2.2. Unmarshaller

Similar to the Marshaller, there is the org.springframework.oxm.Unmarshaller interface.

public interface Unmarshaller {

/\*\*

\* Unmarshal the given provided Source into an object graph.

\*/

Object unmarshal(Source source) throws XmlMappingException, IOException;

}

This interface also has one method, which reads from the given javax.xml.transform.Source (an XML input abstraction), and returns the object read. As with Result, Source is a tagging interface that has three concrete implementations. Each wraps a different XML representation, as indicated in the table below.

Source implementation Wraps XML representation

DOMSource

org.w3c.dom.Node

SAXSource

org.xml.sax.InputSource, and org.xml.sax.XMLReader

StreamSource

java.io.File, java.io.InputStream, or java.io.Reader

Even though there are two separate marshalling interfaces ( Marshaller and Unmarshaller), all implementations found in Spring-WS implement both in one class. This means that you can wire up one marshaller class and refer to it both as a marshaller and an unmarshaller in your applicationContext.xml.

5.2.3. XmlMappingException

Spring converts exceptions from the underlying O/X mapping tool to its own exception hierarchy with the XmlMappingException as the root exception. As can be expected, these runtime exceptions wrap the original exception so no information will be lost.

Additionally, the MarshallingFailureException and UnmarshallingFailureException provide a distinction between marshalling and unmarshalling operations, even though the underlying O/X mapping tool does not do so.

The O/X Mapping exception hierarchy is shown in the following figure:

oxm exceptions

O/X Mapping exception hierarchy

5.3. Using Marshaller and Unmarshaller

Spring’s OXM can be used for a wide variety of situations. In the following example, we will use it to marshal the settings of a Spring-managed application as an XML file. We will use a simple JavaBean to represent the settings:

public class Settings {

private boolean fooEnabled;

public boolean isFooEnabled() {

return fooEnabled;

}

public void setFooEnabled(boolean fooEnabled) {

this.fooEnabled = fooEnabled;

}

}

The application class uses this bean to store its settings. Besides a main method, the class has two methods: saveSettings() saves the settings bean to a file named settings.xml, and loadSettings() loads these settings again. A main() method constructs a Spring application context, and calls these two methods.

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import javax.xml.transform.stream.StreamResult;

import javax.xml.transform.stream.StreamSource;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.springframework.oxm.Marshaller;

import org.springframework.oxm.Unmarshaller;

public class Application {

private static final String FILE\_NAME = "settings.xml";

private Settings settings = new Settings();

private Marshaller marshaller;

private Unmarshaller unmarshaller;

public void setMarshaller(Marshaller marshaller) {

this.marshaller = marshaller;

}

public void setUnmarshaller(Unmarshaller unmarshaller) {

this.unmarshaller = unmarshaller;

}

public void saveSettings() throws IOException {

FileOutputStream os = null;

try {

os = new FileOutputStream(FILE\_NAME);

this.marshaller.marshal(settings, new StreamResult(os));

} finally {

if (os != null) {

os.close();

}

}

}

public void loadSettings() throws IOException {

FileInputStream is = null;

try {

is = new FileInputStream(FILE\_NAME);

this.settings = (Settings) this.unmarshaller.unmarshal(new StreamSource(is));

} finally {

if (is != null) {

is.close();

}

}

}

public static void main(String[] args) throws IOException {

ApplicationContext appContext =

new ClassPathXmlApplicationContext("applicationContext.xml");

Application application = (Application) appContext.getBean("application");

application.saveSettings();

application.loadSettings();

}

}

The Application requires both a marshaller and unmarshaller property to be set. We can do so using the following applicationContext.xml:

<beans>

<bean id="application" class="Application">

<property name="marshaller" ref="castorMarshaller" />

<property name="unmarshaller" ref="castorMarshaller" />

</bean>

<bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"/>

</beans>

This application context uses Castor, but we could have used any of the other marshaller instances described later in this chapter. Note that Castor does not require any further configuration by default, so the bean definition is rather simple. Also note that the CastorMarshaller implements both Marshaller and Unmarshaller, so we can refer to the castorMarshaller bean in both the marshaller and unmarshaller property of the application.

This sample application produces the following settings.xml file:

<?xml version="1.0" encoding="UTF-8"?>

<settings foo-enabled="false"/>

5.4. XML configuration namespace

Marshallers could be configured more concisely using tags from the OXM namespace. To make these tags available, the appropriate schema has to be referenced first in the preamble of the XML configuration file. Note the 'oxm' related text below:

<?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:oxm="http://www.springframework.org/schema/oxm" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd \*http://www.springframework.org/schema/oxm http://www.springframework.org/schema/oxm/spring-oxm.xsd"\*>

Currently, the following tags are available:

jaxb2-marshaller

jibx-marshaller

castor-marshaller

Each tag will be explained in its respective marshaller’s section. As an example though, here is how the configuration of a JAXB2 marshaller might look like:

<oxm:jaxb2-marshaller id="marshaller" contextPath="org.springframework.ws.samples.airline.schema"/>

5.5. JAXB

The JAXB binding compiler translates a W3C XML Schema into one or more Java classes, a jaxb.properties file, and possibly some resource files. JAXB also offers a way to generate a schema from annotated Java classes.

Spring supports the JAXB 2.0 API as XML marshalling strategies, following the Marshaller and Unmarshaller interfaces described in Marshaller and Unmarshaller. The corresponding integration classes reside in the org.springframework.oxm.jaxb package.

5.5.1. Jaxb2Marshaller

The Jaxb2Marshaller class implements both the Spring Marshaller and Unmarshaller interface. It requires a context path to operate, which you can set using the contextPath property. The context path is a list of colon (:) separated Java package names that contain schema derived classes. It also offers a classesToBeBound property, which allows you to set an array of classes to be supported by the marshaller. Schema validation is performed by specifying one or more schema resource to the bean, like so:

<beans>

<bean id="jaxb2Marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">

<property name="classesToBeBound">

<list>

<value>org.springframework.oxm.jaxb.Flight</value>

<value>org.springframework.oxm.jaxb.Flights</value>

</list>

</property>

<property name="schema" value="classpath:org/springframework/oxm/schema.xsd"/>

</bean>

...

</beans>

XML configuration namespace

The jaxb2-marshaller tag configures a org.springframework.oxm.jaxb.Jaxb2Marshaller. Here is an example:

<oxm:jaxb2-marshaller id="marshaller" contextPath="org.springframework.ws.samples.airline.schema"/>

Alternatively, the list of classes to bind can be provided to the marshaller via the class-to-be-bound child tag:

<oxm:jaxb2-marshaller id="marshaller">

<oxm:class-to-be-bound name="org.springframework.ws.samples.airline.schema.Airport"/>

<oxm:class-to-be-bound name="org.springframework.ws.samples.airline.schema.Flight"/>

...

</oxm:jaxb2-marshaller>

Available attributes are:

Attribute Description Required

id

the id of the marshaller

no

contextPath

the JAXB Context path

no

5.6. Castor

Castor XML mapping is an open source XML binding framework. It allows you to transform the data contained in a java object model into/from an XML document. By default, it does not require any further configuration, though a mapping file can be used to have more control over the behavior of Castor.

For more information on Castor, refer to the Castor web site. The Spring integration classes reside in the org.springframework.oxm.castor package.

5.6.1. CastorMarshaller

As with JAXB, the CastorMarshaller implements both the Marshaller and Unmarshaller interface. It can be wired up as follows:

<beans>

<bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller" />

...

</beans>

5.6.2. Mapping

Although it is possible to rely on Castor’s default marshalling behavior, it might be necessary to have more control over it. This can be accomplished using a Castor mapping file. For more information, refer to Castor XML Mapping.

The mapping can be set using the mappingLocation resource property, indicated below with a classpath resource.

<beans>

<bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller" >

<property name="mappingLocation" value="classpath:mapping.xml" />

</bean>

</beans>

XML configuration namespace

The castor-marshaller tag configures a org.springframework.oxm.castor.CastorMarshaller. Here is an example:

<oxm:castor-marshaller id="marshaller" mapping-location="classpath:org/springframework/oxm/castor/mapping.xml"/>

The marshaller instance can be configured in two ways, by specifying either the location of a mapping file (through the mapping-location property), or by identifying Java POJOs (through the target-class or target-package properties) for which there exist corresponding XML descriptor classes. The latter way is usually used in conjunction with XML code generation from XML schemas.

Available attributes are:

Attribute Description Required

id

the id of the marshaller

no

encoding

the encoding to use for unmarshalling from XML

no

target-class

a Java class name for a POJO for which an XML class descriptor is available (as generated through code generation)

no

target-package

a Java package name that identifies a package that contains POJOs and their corresponding Castor XML descriptor classes (as generated through code generation from XML schemas)

no

mapping-location

location of a Castor XML mapping file

no

5.7. JiBX

The JiBX framework offers a solution similar to that which Hibernate provides for ORM: a binding definition defines the rules for how your Java objects are converted to or from XML. After preparing the binding and compiling the classes, a JiBX binding compiler enhances the class files, and adds code to handle converting instances of the classes from or to XML.

For more information on JiBX, refer to the JiBX web site. The Spring integration classes reside in the org.springframework.oxm.jibx package.

5.7.1. JibxMarshaller

The JibxMarshaller class implements both the Marshaller and Unmarshaller interface. To operate, it requires the name of the class to marshal in, which you can set using the targetClass property. Optionally, you can set the binding name using the bindingName property. In the next sample, we bind the Flights class:

<beans>

<bean id="jibxFlightsMarshaller" class="org.springframework.oxm.jibx.JibxMarshaller">

<property name="targetClass">org.springframework.oxm.jibx.Flights</property>

</bean>

...

</beans>

A JibxMarshaller is configured for a single class. If you want to marshal multiple classes, you have to configure multiple JibxMarshallers with different targetClass property values.

XML configuration namespace

The jibx-marshaller tag configures a org.springframework.oxm.jibx.JibxMarshaller. Here is an example:

<oxm:jibx-marshaller id="marshaller" target-class="org.springframework.ws.samples.airline.schema.Flight"/>

Available attributes are:

Attribute Description Required

id

the id of the marshaller

no

target-class

the target class for this marshaller

yes

bindingName

the binding name used by this marshaller

no

5.8. XStream

XStream is a simple library to serialize objects to XML and back again. It does not require any mapping, and generates clean XML.

For more information on XStream, refer to the XStream web site. The Spring integration classes reside in the org.springframework.oxm.xstream package.

5.8.1. XStreamMarshaller

The XStreamMarshaller does not require any configuration, and can be configured in an application context directly. To further customize the XML, you can set analias map, which consists of string aliases mapped to classes:

<beans>

<bean id="xstreamMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller">

<property name="aliases">

<props>

<prop key="Flight">org.springframework.oxm.xstream.Flight</prop>

</props>

</property>

</bean>

...

</beans>

By default, XStream allows for arbitrary classes to be unmarshalled, which can lead to unsafe Java serialization effects. As such, it is not recommended to use the XStreamMarshaller to unmarshal XML from external sources (i.e. the Web), as this can result in security vulnerabilities.

If you choose to use the XStreamMarshaller to unmarshal XML from an external source, set the supportedClasses property on the XStreamMarshaller, like as follows:

<bean id="xstreamMarshaller" class="org.springframework.oxm.xstream.XStreamMarshaller">

<property name="supportedClasses" value="org.springframework.oxm.xstream.Flight"/>

...

</bean>

This will make sure that only the registered classes are eligible for unmarshalling.

Additionally, you can register custom converters to make sure that only your supported classes can be unmarshalled. You might want to add a CatchAllConverter as the last converter in the list, in addition to converters that explicitly support the domain classes that should be supported. As a result, default XStream converters with lower priorities and possible security vulnerabilities do not get invoked.

Note that XStream is an XML serialization library, not a data binding library. Therefore, it has limited namespace support. As such, it is rather unsuitable for usage within Web services.

6. Appendix

6.1. XML Schemas

This part of the appendix lists XML schemas for data access.

6.1.1. The tx schema

The tx tags deal with configuring all of those beans in Spring’s comprehensive support for transactions. These tags are covered in the chapter entitled Transaction Management.

You are strongly encouraged to look at the 'spring-tx.xsd' file that ships with the Spring distribution. This file is (of course), the XML Schema for Spring’s transaction configuration, and covers all of the various tags in the tx namespace, including attribute defaults and suchlike. This file is documented inline, and thus the information is not repeated here in the interests of adhering to the DRY (Don’t Repeat Yourself) principle.

In the interest of completeness, to use the tags in the tx schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the tx namespace are available to you.

<?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:aop="http://www.springframework.org/schema/aop"

xmlns:tx="http://www.springframework.org/schema/tx" xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd"> <!-- bean definitions here -->

</beans>

Often when using the tags in the tx namespace you will also be using the tags from the aop namespace (since the declarative transaction support in Spring is implemented using AOP). The above XML snippet contains the relevant lines needed to reference the aop schema so that the tags in the aop namespace are available to you.

6.1.2. The jdbc schema

The jdbc tags allow you to quickly configure an embedded database or initialize an existing data source. These tags are documented in Embedded database support and Initializing a DataSource respectively.

To use the tags in the jdbc schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the jdbc namespace are available to you.

<?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:jdbc="http://www.springframework.org/schema/jdbc" xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc.xsd"> <!-- bean definitions here -->

</beans>

## Web on Servlet Stack

Version 5.0.8.RELEASE

This part of the documentation covers support for Servlet stack, web applications built on the Servlet API and deployed to Servlet containers. Individual chapters include Spring MVC, View Technologies, CORS Support, and WebSocket Support. For reactive stack, web applications, go to Web on Reactive Stack.

1. Spring Web MVC

1.1. Introduction

Spring Web MVC is the original web framework built on the Servlet API and included in the Spring Framework from the very beginning. The formal name "Spring Web MVC" comes from the name of its source module spring-webmvc but it is more commonly known as "Spring MVC".

Parallel to Spring Web MVC, Spring Framework 5.0 introduced a reactive stack, web framework whose name Spring WebFlux is also based on its source module spring-webflux. This section covers Spring Web MVC. The next section covers Spring WebFlux.

For baseline information and compatibility with Servlet container and Java EE version ranges please visit the Spring Framework Wiki.

1.2. DispatcherServlet

Same in Spring WebFlux

Spring MVC, like many other web frameworks, is designed around the front controller pattern where a central Servlet, the DispatcherServlet, provides a shared algorithm for request processing while actual work is performed by configurable, delegate components. This model is flexible and supports diverse workflows.

The DispatcherServlet, as any Servlet, needs to be declared and mapped according to the Servlet specification using Java configuration or in web.xml. In turn the DispatcherServlet uses Spring configuration to discover the delegate components it needs for request mapping, view resolution, exception handling, and more.

Below is an example of the Java configuration that registers and initializes the DispatcherServlet. This class is auto-detected by the Servlet container (see Servlet Config):

public class MyWebApplicationInitializer implements WebApplicationInitializer {

@Override

public void onStartup(ServletContext servletCxt) {

// Load Spring web application configuration

AnnotationConfigWebApplicationContext ac = new AnnotationConfigWebApplicationContext();

ac.register(AppConfig.class);

ac.refresh();

// Create and register the DispatcherServlet

DispatcherServlet servlet = new DispatcherServlet(ac);

ServletRegistration.Dynamic registration = servletCxt.addServlet("app", servlet);

registration.setLoadOnStartup(1);

registration.addMapping("/app/\*");

}

}

In addition to using the ServletContext API directly, you can also extend AbstractAnnotationConfigDispatcherServletInitializer and override specific methods (see example under Context Hierarchy).

Below is an example of web.xml configuration to register and initialize the DispatcherServlet:

<web-app>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/app-context.xml</param-value>

</context-param>

<servlet>

<servlet-name>app</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value></param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>app</servlet-name>

<url-pattern>/app/\*</url-pattern>

</servlet-mapping>

</web-app>

Spring Boot follows a different initialization sequence. Rather than hooking into the lifecycle of the Servlet container, Spring Boot uses Spring configuration to bootstrap itself and the embedded Servlet container. Filter and Servlet declarations are detected in Spring configuration and registered with the Servlet container. For more details check the Spring Boot docs.

1.2.1. Context Hierarchy

DispatcherServlet expects a WebApplicationContext, an extension of a plain ApplicationContext, for its own configuration. WebApplicationContext has a link to the ServletContext and Servlet it is associated with. It is also bound to the ServletContext such that applications can use static methods on RequestContextUtils to look up the WebApplicationContext if they need access to it.

For many applications having a single WebApplicationContext is simple and sufficient. It is also possible to have a context hierarchy where one root WebApplicationContext is shared across multiple DispatcherServlet (or other Servlet) instances, each with its own child WebApplicationContext configuration. See Additional Capabilities of the ApplicationContext for more on the context hierarchy feature.

The root WebApplicationContext typically contains infrastructure beans such as data repositories and business services that need to be shared across multiple Servlet instances. Those beans are effectively inherited and could be overridden (i.e. re-declared) in the Servlet-specific, child WebApplicationContext which typically contains beans local to the given Servlet:

mvc context hierarchy

Below is example configuration with a WebApplicationContext hierarchy:

public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

@Override

protected Class<?>[] getRootConfigClasses() {

return new Class<?>[] { RootConfig.class };

}

@Override

protected Class<?>[] getServletConfigClasses() {

return new Class<?>[] { App1Config.class };

}

@Override

protected String[] getServletMappings() {

return new String[] { "/app1/\*" };

}

}

If an application context hierarchy is not required, applications may return all configuration via getRootConfigClasses() and null from getServletConfigClasses().

And the web.xml equivalent:

<web-app>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/root-context.xml</param-value>

</context-param>

<servlet>

<servlet-name>app1</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/app1-context.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>app1</servlet-name>

<url-pattern>/app1/\*</url-pattern>

</servlet-mapping>

</web-app>

If an application context hierarchy is not required, applications may configure a "root" context only and leave the contextConfigLocation Servlet parameter empty.

1.2.2. Special Bean Types

Same in Spring WebFlux

The DispatcherServlet delegates to special beans to process requests and render the appropriate responses. By "special beans" we mean Spring-managed, Object instances that implement WebFlux framework contracts. Those usually come with built-in contracts but you can customize their properties, extend or replace them.

The table below lists the special beans detected by the DispatcherHandler:

Bean type Explanation

HandlerMapping

Map a request to a handler along with a list of interceptors for pre- and post-processing. The mapping is based on some criteria the details of which vary by HandlerMapping implementation.

The two main HandlerMapping implementations are RequestMappingHandlerMapping which supports @RequestMapping annotated methods and SimpleUrlHandlerMapping which maintains explicit registrations of URI path patterns to handlers.

HandlerAdapter

Help the DispatcherServlet to invoke a handler mapped to a request regardless of how the handler is actually invoked. For example, invoking an annotated controller requires resolving annotations. The main purpose of a HandlerAdapter is to shield the DispatcherServlet from such details.

HandlerExceptionResolver

Strategy to resolve exceptions possibly mapping them to handlers, or to HTML error views, or other. See Exceptions.

ViewResolver

Resolve logical String-based view names returned from a handler to an actual View to render to the response with. See View Resolution and View Technologies.

LocaleResolver, LocaleContextResolver

Resolve the Locale a client is using and possibly their time zone, in order to be able to offer internationalized views. See Locale.

ThemeResolver

Resolve themes your web application can use, for example, to offer personalized layouts. See Themes.

MultipartResolver

Abstraction for parsing a multi-part request (e.g. browser form file upload) with the help of some multipart parsing library. See Multipart resolver.

FlashMapManager

Store and retrieve the "input" and the "output" FlashMap that can be used to pass attributes from one request to another, usually across a redirect. See Flash attributes.

1.2.3. Web MVC Config

Same in Spring WebFlux

Applications can declare the infrastructure beans listed in Special Bean Types that are required to process requests. The DispatcherServlet checks the WebApplicationContext for each special bean. If there are no matching bean types, it falls back on the default types listed in DispatcherServlet.properties.

In most cases the MVC Config is the best starting point. It declares the required beans in either Java or XML, and provides a higher level configuration callback API to customize it.

Spring Boot relies on the MVC Java config to configure Spring MVC and also provides many extra convenient options.

1.2.4. Servlet Config

In a Servlet 3.0+ environment, you have the option of configuring the Servlet container programmatically as an alternative or in combination with a web.xml file. Below is an example of registering a DispatcherServlet:

import org.springframework.web.WebApplicationInitializer;

public class MyWebApplicationInitializer implements WebApplicationInitializer {

@Override

public void onStartup(ServletContext container) {

XmlWebApplicationContext appContext = new XmlWebApplicationContext();

appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new DispatcherServlet(appContext));

registration.setLoadOnStartup(1);

registration.addMapping("/");

}

}

WebApplicationInitializer is an interface provided by Spring MVC that ensures your implementation is detected and automatically used to initialize any Servlet 3 container. An abstract base class implementation of WebApplicationInitializer named AbstractDispatcherServletInitializer makes it even easier to register the DispatcherServlet by simply overriding methods to specify the servlet mapping and the location of the DispatcherServlet configuration.

This is recommended for applications that use Java-based Spring configuration:

public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

@Override

protected Class<?>[] getRootConfigClasses() {

return null;

}

@Override

protected Class<?>[] getServletConfigClasses() {

return new Class<?>[] { MyWebConfig.class };

}

@Override

protected String[] getServletMappings() {

return new String[] { "/" };

}

}

If using XML-based Spring configuration, you should extend directly from AbstractDispatcherServletInitializer:

public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

@Override

protected WebApplicationContext createRootApplicationContext() {

return null;

}

@Override

protected WebApplicationContext createServletApplicationContext() {

XmlWebApplicationContext cxt = new XmlWebApplicationContext();

cxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

return cxt;

}

@Override

protected String[] getServletMappings() {

return new String[] { "/" };

}

}

AbstractDispatcherServletInitializer also provides a convenient way to add Filter instances and have them automatically mapped to the DispatcherServlet:

public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

// ...

@Override

protected Filter[] getServletFilters() {

return new Filter[] {

new HiddenHttpMethodFilter(), new CharacterEncodingFilter() };

}

}

Each filter is added with a default name based on its concrete type and automatically mapped to the DispatcherServlet.

The isAsyncSupported protected method of AbstractDispatcherServletInitializer provides a single place to enable async support on the DispatcherServlet and all filters mapped to it. By default this flag is set to true.

Finally, if you need to further customize the DispatcherServlet itself, you can override the createDispatcherServlet method.

1.2.5. Processing

Same in Spring WebFlux

The DispatcherServlet processes requests as follows:

The WebApplicationContext is searched for and bound in the request as an attribute that the controller and other elements in the process can use. It is bound by default under the key DispatcherServlet.WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE.

The locale resolver is bound to the request to enable elements in the process to resolve the locale to use when processing the request (rendering the view, preparing data, and so on). If you do not need locale resolving, you do not need it.

The theme resolver is bound to the request to let elements such as views determine which theme to use. If you do not use themes, you can ignore it.

If you specify a multipart file resolver, the request is inspected for multiparts; if multiparts are found, the request is wrapped in a MultipartHttpServletRequest for further processing by other elements in the process. See Multipart resolver for further information about multipart handling.

An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) is executed in order to prepare a model or rendering. Or alternatively for annotated controllers, the response may be rendered (within the HandlerAdapter) instead of returning a view.

If a model is returned, the view is rendered. If no model is returned, (may be due to a preprocessor or postprocessor intercepting the request, perhaps for security reasons), no view is rendered, because the request could already have been fulfilled.

The HandlerExceptionResolver beans declared in the WebApplicationContext are used to resolve exceptions thrown during request processing. Those exception resolvers allow customizing the logic to address exceptions. See Exceptions for more details.

The Spring DispatcherServlet also supports the return of the last-modification-date, as specified by the Servlet API. The process of determining the last modification date for a specific request is straightforward: the DispatcherServlet looks up an appropriate handler mapping and tests whether the handler that is found implements the LastModified interface. If so, the value of the long getLastModified(request) method of the LastModified interface is returned to the client.

You can customize individual DispatcherServlet instances by adding Servlet initialization parameters ( init-param elements) to the Servlet declaration in the web.xml file. See the following table for the list of supported parameters.

Table 1. DispatcherServlet initialization parameters

Parameter Explanation

contextClass

Class that implements WebApplicationContext, which instantiates the context used by this Servlet. By default, the XmlWebApplicationContext is used.

contextConfigLocation

String that is passed to the context instance (specified by contextClass) to indicate where context(s) can be found. The string consists potentially of multiple strings (using a comma as a delimiter) to support multiple contexts. In case of multiple context locations with beans that are defined twice, the latest location takes precedence.

namespace

Namespace of the WebApplicationContext. Defaults to [servlet-name]-servlet.

throwExceptionIfNoHandlerFound

Whether to throw a NoHandlerFoundException when no handler was found for a request. The exception can then be caught with a HandlerExceptionResolver, e.g. via an @ExceptionHandler controller method, and handled as any others.

By default this is set to "false", in which case the DispatcherServlet sets the response status to 404 (NOT\_FOUND) without raising an exception.

Note that if default servlet handling is also configured, then unresolved requests are always forwarded to the default servlet and a 404 would never be raised.

1.2.6. Interception

All HandlerMapping implementations supports handler interceptors that are useful when you want to apply specific functionality to certain requests, for example, checking for a principal. Interceptors must implement HandlerInterceptor from the org.springframework.web.servlet package with three methods that should provide enough flexibility to do all kinds of pre-processing and post-processing:

preHandle(..) — before the actual handler is executed

postHandle(..) — after the handler is executed

afterCompletion(..) — after the complete request has finished

The preHandle(..) method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns true, the handler execution chain will continue; when it returns false, the DispatcherServlet assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain.

See Interceptors in the section on MVC configuration for examples of how to configure interceptors. You can also register them directly via setters on individual HandlerMapping implementations.

Note that postHandle is less useful with @ResponseBody and ResponseEntity methods for which the response is written and committed within the HandlerAdapter and before postHandle. That means its too late to make any changes to the response such as adding an extra header. For such scenarios you can implement ResponseBodyAdvice and either declare it as an Controller Advice bean or configure it directly on RequestMappingHandlerAdapter.

1.2.7. Exceptions

Same in Spring WebFlux

If an exception occurs during request mapping or is thrown from a request handler such as an @Controller, the DispatcherServlet delegates to a chain of HandlerExceptionResolver beans to resolve the exception and provide alternative handling, which typically is an error response.

The table below lists the available HandlerExceptionResolver implementations:

Table 2. HandlerExceptionResolver implementations

HandlerExceptionResolver Description

SimpleMappingExceptionResolver

A mapping between exception class names and error view names. Useful for rendering error pages in a browser application.

DefaultHandlerExceptionResolver

Resolves exceptions raised by Spring MVC and maps them to HTTP status codes. Also see alternative ResponseEntityExceptionHandler and REST API exceptions.

ResponseStatusExceptionResolver

Resolves exceptions with the @ResponseStatus annotation and maps them to HTTP status codes based on the value in the annotation.

ExceptionHandlerExceptionResolver

Resolves exceptions by invoking an @ExceptionHandler method in an @Controller or an @ControllerAdvice class. See @ExceptionHandler methods.

Chain of resolvers

You can form an exception resolver chain simply by declaring multiple HandlerExceptionResolver beans in your Spring configuration and setting their order properties as needed. The higher the order property, the later the exception resolver is positioned.

The contract of HandlerExceptionResolver specifies that it can return:

ModelAndView that points to an error view.

Empty ModelAndView if the exception was handled within the resolver.

null if the exception remains unresolved, for subsequent resolvers to try; and if the exception remains at the end, it is allowed to bubble up to the Servlet container.

The MVC Config automatically declares built-in resolvers for default Spring MVC exceptions, for @ResponseStatus annotated exceptions, and for support of @ExceptionHandler methods. You can customize that list or replace it.

Container error page

If an exception remains unresolved by any HandlerExceptionResolver and is therefore left to propagate, or if the response status is set to an error status (i.e. 4xx, 5xx), Servlet containers may render a default error page in HTML. To customize the default error page of the container, you can declare an error page mapping in web.xml:

<error-page>

<location>/error</location>

</error-page>

Given the above, when an exception bubbles up, or the response has an error status, the Servlet container makes an ERROR dispatch within the container to the configured URL (e.g. "/error"). This is then processed by the DispatcherServlet, possibly mapping it to an @Controller which could be implemented to return an error view name with a model or to render a JSON response as shown below:

@RestController

public class ErrorController {

@RequestMapping(path = "/error")

public Map<String, Object> handle(HttpServletRequest request) {

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

map.put("status", request.getAttribute("javax.servlet.error.status\_code"));

map.put("reason", request.getAttribute("javax.servlet.error.message"));

return map;

}

}

The Servlet API does not provide a way to create error page mappings in Java. You can however use both an WebApplicationInitializer and a minimal web.xml.

1.2.8. View Resolution

Same in Spring WebFlux

Spring MVC defines the ViewResolver and View interfaces that enable you to render models in a browser without tying you to a specific view technology. ViewResolver provides a mapping between view names and actual views. View addresses the preparation of data before handing over to a specific view technology.

The table below provides more details on the ViewResolver hierarchy:

Table 3. ViewResolver implementations

ViewResolver Description

AbstractCachingViewResolver

Sub-classes of AbstractCachingViewResolver cache view instances that they resolve. Caching improves performance of certain view technologies. It’s possible to turn off the cache by setting the cache property to false. Furthermore, if you must refresh a certain view at runtime (for example when a FreeMarker template is modified), you can use the removeFromCache(String viewName, Locale loc) method.

XmlViewResolver

Implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring’s XML bean factories. The default configuration file is /WEB-INF/views.xml.

ResourceBundleViewResolver

Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle base name, and for each view it is supposed to resolve, it uses the value of the property [viewname].(class) as the view class and the value of the property [viewname].url as the view url. Examples can be found in the chapter on View Technologies.

UrlBasedViewResolver

Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.

InternalResourceViewResolver

Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..). See the UrlBasedViewResolver javadocs for details.

FreeMarkerViewResolver

Convenient subclass of UrlBasedViewResolver that supports FreeMarkerView and custom subclasses of them.

ContentNegotiatingViewResolver

Implementation of the ViewResolver interface that resolves a view based on the request file name or Accept header. See Content negotiation.

Handling

Same in Spring WebFlux

You chain view resolvers by declaring more than one resolver beans and, if necessary, by setting the order property to specify ordering. Remember, the higher the order property, the later the view resolver is positioned in the chain.

The contract of a ViewResolver specifies that it can return null to indicate the view could not be found. However in the case of JSPs, and InternalResourceViewResolver, the only way to figure out if a JSP exists is to perform a dispatch through RequestDispatcher. Therefore an InternalResourceViewResolver must always be configured to be last in the overall order of view resolvers.

To configure view resolution is as simple as adding ViewResolver beans to your Spring configuration. The MVC Config provides provides a dedicated configuration API for View Resolvers and also for adding logic-less View Controllers which are useful for HTML template rendering without controller logic.

Redirecting

Same in Spring WebFlux

The special redirect: prefix in a view name allows you to perform a redirect. The UrlBasedViewResolver (and sub-classes) recognize this as an instruction that a redirect is needed. The rest of the view name is the redirect URL.

The net effect is the same as if the controller had returned a RedirectView, but now the controller itself can simply operate in terms of logical view names. A logical view name such as redirect:/myapp/some/resource will redirect relative to the current Servlet context, while a name such as redirect:http://myhost.com/some/arbitrary/path will redirect to an absolute URL.

Note that if a controller method is annotated with the @ResponseStatus, the annotation value takes precedence over the response status set by RedirectView.

Forwarding

It is also possible to use a special forward: prefix for view names that are ultimately resolved by UrlBasedViewResolver and subclasses. This creates an InternalResourceView which does a RequestDispatcher.forward(). Therefore, this prefix is not useful with InternalResourceViewResolver and InternalResourceView (for JSPs) but it can be helpful if using another view technology, but still want to force a forward of a resource to be handled by the Servlet/JSP engine. Note that you may also chain multiple view resolvers, instead.

Content negotiation

Same in Spring WebFlux

ContentNegotiatingViewResolver does not resolve views itself but rather delegates to other view resolvers, and selects the view that resembles the representation requested by the client. The representation can be determined from the Accept header or from a query parameter, e.g. "/path?format=pdf".

The ContentNegotiatingViewResolver selects an appropriate View to handle the request by comparing the request media type(s) with the media type (also known as Content-Type) supported by the View associated with each of its ViewResolvers. The first View in the list that has a compatible Content-Type returns the representation to the client. If a compatible view cannot be supplied by the ViewResolver chain, then the list of views specified through the DefaultViews property will be consulted. This latter option is appropriate for singleton Views that can render an appropriate representation of the current resource regardless of the logical view name. The Accept header may include wild cards, for example text/\*, in which case a View whose Content-Type was text/xml is a compatible match.

See View Resolvers under MVC Config for configuration details.

1.2.9. Locale

Most parts of Spring’s architecture support internationalization, just as the Spring web MVC framework does. DispatcherServlet enables you to automatically resolve messages using the client’s locale. This is done with LocaleResolver objects.

When a request comes in, the DispatcherServlet looks for a locale resolver, and if it finds one it tries to use it to set the locale. Using the RequestContext.getLocale() method, you can always retrieve the locale that was resolved by the locale resolver.

In addition to automatic locale resolution, you can also attach an interceptor to the handler mapping (see Interception for more information on handler mapping interceptors) to change the locale under specific circumstances, for example, based on a parameter in the request.

Locale resolvers and interceptors are defined in the org.springframework.web.servlet.i18n package and are configured in your application context in the normal way. Here is a selection of the locale resolvers included in Spring.

TimeZone

In addition to obtaining the client’s locale, it is often useful to know their time zone. The LocaleContextResolver interface offers an extension to LocaleResolver that allows resolvers to provide a richer LocaleContext, which may include time zone information.

When available, the user’s TimeZone can be obtained using the RequestContext.getTimeZone() method. Time zone information will automatically be used by Date/Time Converter and Formatter objects registered with Spring’s ConversionService.

Header resolver

This locale resolver inspects the accept-language header in the request that was sent by the client (e.g., a web browser). Usually this header field contains the locale of the client’s operating system. Note that this resolver does not support time zone information.

Cookie resolver

This locale resolver inspects a Cookie that might exist on the client to see if a Locale or TimeZone is specified. If so, it uses the specified details. Using the properties of this locale resolver, you can specify the name of the cookie as well as the maximum age. Find below an example of defining a CookieLocaleResolver.

<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

<property name="cookieName" value="clientlanguage"/>

<!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->

<property name="cookieMaxAge" value="100000"/>

</bean>

Table 4. CookieLocaleResolver properties

Property Default Description

cookieName

classname + LOCALE

The name of the cookie

cookieMaxAge

Servlet container default

The maximum time a cookie will stay persistent on the client. If -1 is specified, the cookie will not be persisted; it will only be available until the client shuts down their browser.

cookiePath

/

Limits the visibility of the cookie to a certain part of your site. When cookiePath is specified, the cookie will only be visible to that path and the paths below it.

Session resolver

The SessionLocaleResolver allows you to retrieve Locale and TimeZone from the session that might be associated with the user’s request. In contrast to CookieLocaleResolver, this strategy stores locally chosen locale settings in the Servlet container’s HttpSession. As a consequence, those settings are just temporary for each session and therefore lost when each session terminates.

Note that there is no direct relationship with external session management mechanisms such as the Spring Session project. This SessionLocaleResolver will simply evaluate and modify corresponding HttpSession attributes against the current HttpServletRequest.

Locale interceptor

You can enable changing of locales by adding the LocaleChangeInterceptor to one of the handler mappings (see [mvc-handlermapping]). It will detect a parameter in the request and change the locale. It calls setLocale() on the LocaleResolver that also exists in the context. The following example shows that calls to all \*.view resources containing a parameter named siteLanguage will now change the locale. So, for example, a request for the following URL, http://www.sf.net/home.view?siteLanguage=nl will change the site language to Dutch.

<bean id="localeChangeInterceptor"

class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">

<property name="paramName" value="siteLanguage"/>

</bean>

<bean id="localeResolver"

class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="interceptors">

<list>

<ref bean="localeChangeInterceptor"/>

</list>

</property>

<property name="mappings">

<value>/\*\*/\*.view=someController</value>

</property>

</bean>

1.2.10. Themes

You can apply Spring Web MVC framework themes to set the overall look-and-feel of your application, thereby enhancing user experience. A theme is a collection of static resources, typically style sheets and images, that affect the visual style of the application.

Define a theme

To use themes in your web application, you must set up an implementation of the org.springframework.ui.context.ThemeSource interface. The WebApplicationContext interface extends ThemeSource but delegates its responsibilities to a dedicated implementation. By default the delegate will be an org.springframework.ui.context.support.ResourceBundleThemeSource implementation that loads properties files from the root of the classpath. To use a custom ThemeSource implementation or to configure the base name prefix of the ResourceBundleThemeSource, you can register a bean in the application context with the reserved name themeSource. The web application context automatically detects a bean with that name and uses it.

When using the ResourceBundleThemeSource, a theme is defined in a simple properties file. The properties file lists the resources that make up the theme. Here is an example:

styleSheet=/themes/cool/style.css

background=/themes/cool/img/coolBg.jpg

The keys of the properties are the names that refer to the themed elements from view code. For a JSP, you typically do this using the spring:theme custom tag, which is very similar to the spring:message tag. The following JSP fragment uses the theme defined in the previous example to customize the look and feel:

<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>

<html>

<head>

<link rel="stylesheet" href="<spring:theme code='styleSheet'/>" type="text/css"/>

</head>

<body style="background=<spring:theme code='background'/>">

...

</body>

</html>

By default, the ResourceBundleThemeSource uses an empty base name prefix. As a result, the properties files are loaded from the root of the classpath. Thus you would put the cool.properties theme definition in a directory at the root of the classpath, for example, in /WEB-INF/classes. The ResourceBundleThemeSource uses the standard Java resource bundle loading mechanism, allowing for full internationalization of themes. For example, we could have a /WEB-INF/classes/cool\_nl.properties that references a special background image with Dutch text on it.

Resolve themes

After you define themes, as in the preceding section, you decide which theme to use. The DispatcherServlet will look for a bean named themeResolver to find out which ThemeResolver implementation to use. A theme resolver works in much the same way as a LocaleResolver. It detects the theme to use for a particular request and can also alter the request’s theme. The following theme resolvers are provided by Spring:

Table 5. ThemeResolver implementations

Class Description

FixedThemeResolver

Selects a fixed theme, set using the defaultThemeName property.

SessionThemeResolver

The theme is maintained in the user’s HTTP session. It only needs to be set once for each session, but is not persisted between sessions.

CookieThemeResolver

The selected theme is stored in a cookie on the client.

Spring also provides a ThemeChangeInterceptor that allows theme changes on every request with a simple request parameter.

1.2.11. Multipart resolver

Same in Spring WebFlux

MultipartResolver from the org.springframework.web.multipart package is a strategy for parsing multipart requests including file uploads. There is one implementation based on Commons FileUpload and another based on Servlet 3.0 multipart request parsing.

To enable multipart handling, you need declare a MultipartResolver bean in your DispatcherServlet Spring configuration with the name "multipartResolver". The DispatcherServlet detects it and applies it to incoming request. When a POST with content-type of "multipart/form-data" is received, the resolver parses the content and wraps the current HttpServletRequest as MultipartHttpServletRequest in order to provide access to resolved parts in addition to exposing them as request parameters.

Apache FileUpload

To use Apache Commons FileUpload, simply configure a bean of type CommonsMultipartResolver with the name multipartResolver. Of course you also need to have commons-fileupload as a dependency on your classpath.

Servlet 3.0

Servlet 3.0 multipart parsing needs to be enabled through Servlet container configuration:

in Java, set a MultipartConfigElement on the Servlet registration.

in web.xml, add a "<multipart-config>" section to the servlet declaration.

public class AppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

// ...

@Override

protected void customizeRegistration(ServletRegistration.Dynamic registration) {

// Optionally also set maxFileSize, maxRequestSize, fileSizeThreshold

registration.setMultipartConfig(new MultipartConfigElement("/tmp"));

}

}

Once the Servlet 3.0 configuration is in place, simply add a bean of type StandardServletMultipartResolver with the name multipartResolver.

1.3. Filters

Same in Spring WebFlux

The spring-web module provides some useful filters.

1.3.1. HTTP PUT Form

Browsers can only submit form data via HTTP GET or HTTP POST but non-browser clients can also use HTTP PUT and PATCH. The Servlet API requires ServletRequest.getParameter\*() methods to support form field access only for HTTP POST.

The spring-web module provides HttpPutFormContentFilter that intercepts HTTP PUT and PATCH requests with content type application/x-www-form-urlencoded, reads the form data from the body of the request, and wraps the ServletRequest in order to make the form data available through the ServletRequest.getParameter\*() family of methods.

1.3.2. Forwarded Headers

Same in Spring WebFlux

As a request goes through proxies such as load balancers the host, port, and scheme may change presenting a challenge for applications that need to create links to resources since the links should reflect the host, port, and scheme of the original request as seen from a client perspective.

RFC 7239 defines the "Forwarded" HTTP header for proxies to use to provide information about the original request. There are also other non-standard headers in use such as "X-Forwarded-Host", "X-Forwarded-Port", and "X-Forwarded-Proto".

ForwardedHeaderFilter detects, extracts, and uses information from the "Forwarded" header, or from "X-Forwarded-Host", "X-Forwarded-Port", and "X-Forwarded-Proto". It wraps the request in order to overlay its host, port, and scheme and also "hides" the forwarded headers for subsequent processing.

Note that there are security considerations when using forwarded headers as explained in Section 8 of RFC 7239. At the application level it is difficult to determine whether forwarded headers can be trusted or not. This is why the network upstream should be configured correctly to filter out untrusted forwarded headers from the outside.

Applications that don’t have a proxy and don’t need to use forwarded headers can configure the ForwardedHeaderFilter to remove and ignore such headers.

1.3.3. Shallow ETag

The ShallowEtagHeaderFilter filter creates a "shallow" ETag by caching the content written to the response, and computing an MD5 hash from it. The next time a client sends, it does the same, but also compares the computed value against the If-None-Match request header and if the two are equal, it returns a 304 (NOT\_MODIFIED).

This strategy saves network bandwidth but not CPU, as the full response must be computed for each request. Other strategies at the controller level, described above, can avoid the computation. See HTTP Caching.

This filter has a writeWeakETag parameter that configures the filter to write Weak ETags, like this: W/"02a2d595e6ed9a0b24f027f2b63b134d6", as defined in RFC 7232 Section 2.3.

1.3.4. CORS

Same in Spring WebFlux

Spring MVC provides fine-grained support for CORS configuration through annotations on controllers. However when used with Spring Security it is advisable to rely on the built-in CorsFilter that must be ordered ahead of Spring Security’s chain of filters.

See the section on CORS and the CORS Filter for more details.

1.4. Annotated Controllers

Same in Spring WebFlux

Spring MVC provides an annotation-based programming model where @Controller and @RestController components use annotations to express request mappings, request input, exception handling, and more. Annotated controllers have flexible method signatures and do not have to extend base classes nor implement specific interfaces.

@Controller

public class HelloController {

@GetMapping("/hello")

public String handle(Model model) {

model.addAttribute("message", "Hello World!");

return "index";

}

}

In this particular example the method accepts a Model and returns a view name as a String but many other options exist and are explained further below in this chapter.

Guides and tutorials on spring.io use the annotation-based programming model described in this section.

1.4.1. Declaration

Same in Spring WebFlux

You can define controller beans using a standard Spring bean definition in the Servlet’s WebApplicationContext. The @Controller stereotype allows for auto-detection, aligned with Spring general support for detecting @Component classes in the classpath and auto-registering bean definitions for them. It also acts as a stereotype for the annotated class, indicating its role as a web component.

To enable auto-detection of such @Controller beans, you can add component scanning to your Java configuration:

@Configuration

@ComponentScan("org.example.web")

public class WebConfig {

// ...

}

The XML configuration equivalent:

<?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:p="http://www.springframework.org/schema/p"

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="org.example.web"/>

<!-- ... -->

</beans>

@RestController is a composed annotation that is itself meta-annotated with @Controller and @ResponseBody indicating a controller whose every method inherits the type-level @ResponseBody annotation and therefore writes directly to the response body vs view resolution and rendering with an HTML template.

AOP proxies

In some cases a controller may need to be decorated with an AOP proxy at runtime. One example is if you choose to have @Transactional annotations directly on the controller. When this is the case, for controllers specifically, we recommend using class-based proxying. This is typically the default choice with controllers. However if a controller must implement an interface that is not a Spring Context callback (e.g. InitializingBean, \*Aware, etc), you may need to explicitly configure class-based proxying. For example with <tx:annotation-driven/>, change to <tx:annotation-driven proxy-target-class="true"/>.

1.4.2. Request Mapping

Same in Spring WebFlux

The @RequestMapping annotation is used to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types. It can be used at the class-level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.

There are also HTTP method specific shortcut variants of @RequestMapping:

@GetMapping

@PostMapping

@PutMapping

@DeleteMapping

@PatchMapping

The above are Custom Annotations that are provided out of the box because arguably most controller methods should be mapped to a specific HTTP method vs using @RequestMapping which by default matches to all HTTP methods. At the same an @RequestMapping is still needed at the class level to express shared mappings.

Below is an example with type and method level mappings:

@RestController

@RequestMapping("/persons")

class PersonController {

@GetMapping("/{id}")

public Person getPerson(@PathVariable Long id) {

// ...

}

@PostMapping

@ResponseStatus(HttpStatus.CREATED)

public void add(@RequestBody Person person) {

// ...

}

}

URI patterns

Same in Spring WebFlux

You can map requests using glob patterns and wildcards:

? matches one character

\* matches zero or more characters within a path segment

\*\* match zero or more path segments

You can also declare URI variables and access their values with @PathVariable:

@GetMapping("/owners/{ownerId}/pets/{petId}")

public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {

// ...

}

URI variables can be declared at the class and method level:

@Controller

@RequestMapping("/owners/{ownerId}")

public class OwnerController {

@GetMapping("/pets/{petId}")

public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {

// ...

}

}

URI variables are automatically converted to the appropriate type or`TypeMismatchException` is raised. Simple types — int, long, Date, are supported by default and you can register support for any other data type. See Type Conversion and DataBinder.

URI variables can be named explicitly — e.g. @PathVariable("customId"), but you can leave that detail out if the names are the same and your code is compiled with debugging information or with the -parameters compiler flag on Java 8.

The syntax {varName:regex} declares a URI variable with a regular expressions with the syntax {varName:regex} — e.g. given URL "/spring-web-3.0.5 .jar", the below method extracts the name, version, and file extension:

@GetMapping("/{name:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{ext:\\.[a-z]+}")

public void handle(@PathVariable String version, @PathVariable String ext) {

// ...

}

URI path patterns can also have embedded ${…​} placeholders that are resolved on startup via PropertyPlaceHolderConfigurer against local, system, environment, and other property sources. This can be used for example to parameterize a base URL based on some external configuration.

Spring MVC uses the PathMatcher contract and the AntPathMatcher implementation from spring-core for URI path matching.

Pattern comparison

Same in Spring WebFlux

When multiple patterns match a URL, they must be compared to find the best match. This done via AntPathMatcher.getPatternComparator(String path) which looks for patterns that more specific.

A pattern is less specific if it has a lower count of URI variables and single wildcards counted as 1 and double wildcards counted as 2. Given an equal score, the longer pattern is chosen. Given the same score and length, the pattern with more URI variables than wildcards is chosen.

The default mapping pattern /\*\* is excluded from scoring and always sorted last. Also prefix patterns such as /public/\*\* are considered less specific than other pattern that don’t have double wildcards.

For the full details see AntPatternComparator in AntPathMatcher and also keep mind that the PathMatcher implementation used can be customized. See Path Matching in the configuration section.

Suffix match

By default Spring MVC performs ".\*" suffix pattern matching so that a controller mapped to /person is also implicitly mapped to /person.\*. The file extension is then used to interpret the requested content type to use for the response (i.e. instead of the "Accept" header), e.g. /person.pdf, /person.xml, etc.

Using file extensions like this was necessary when browsers used to send Accept headers that were hard to interpret consistently. At present that is no longer a necessity and using the "Accept" header should be the preferred choice.

Over time the use of file name extensions has proven problematic in a variety of ways. It can cause ambiguity when overlayed with the use of URI variables, path parameters, URI encoding, and it also makes it difficult to reason about URL-based authorization and security (see next section for more details).

To completely disable the use of file extensions, you must set both of these:

useSuffixPatternMatching(false), see PathMatchConfigurer

favorPathExtension(false), see ContentNeogiationConfigurer

URL-based content negotiation can still be useful, for example when typing a URL in a browser. To enable that we recommend a query parameter based strategy to avoid most of the issues that come with file extensions. Or if you must use file extensions, consider restricting them to a list of explicitly registered extensions through the mediaTypes property of ContentNeogiationConfigurer.

Suffix match and RFD

Reflected file download (RFD) attack is similar to XSS in that it relies on request input, e.g. query parameter, URI variable, being reflected in the response. However instead of inserting JavaScript into HTML, an RFD attack relies on the browser switching to perform a download and treating the response as an executable script when double-clicked later.

In Spring MVC @ResponseBody and ResponseEntity methods are at risk because they can render different content types which clients can request via URL path extensions. Disabling suffix pattern matching and the use of path extensions for content negotiation lower the risk but are not sufficient to prevent RFD attacks.

To prevent RFD attacks, prior to rendering the response body Spring MVC adds a Content-Disposition:inline;filename=f.txt header to suggest a fixed and safe download file. This is done only if the URL path contains a file extension that is neither whitelisted nor explicitly registered for content negotiation purposes. However it may potentially have side effects when URLs are typed directly into a browser.

Many common path extensions are whitelisted by default. Applications with custom HttpMessageConverter implementations can explicitly register file extensions for content negotiation to avoid having a Content-Disposition header added for those extensions. See Content Types.

Check CVE-2015-5211 for additional recommendations related to RFD.

Consumable media types

Same in Spring WebFlux

You can narrow the request mapping based on the Content-Type of the request:

@PostMapping(path = "/pets", consumes = "application/json")

public void addPet(@RequestBody Pet pet) {

// ...

}

The consumes attribute also supports negation expressions — e.g. !text/plain means any content type other than "text/plain".

You can declare a shared consumes attribute at the class level. Unlike most other request mapping attributes however when used at the class level, a method-level consumes attribute will overrides rather than extend the class level declaration.

MediaType provides constants for commonly used media types — e.g. APPLICATION\_JSON\_VALUE, APPLICATION\_XML\_VALUE.

Producible media types

Same in Spring WebFlux

You can narrow the request mapping based on the Accept request header and the list of content types that a controller method produces:

@GetMapping(path = "/pets/{petId}", produces = "application/json;charset=UTF-8")

@ResponseBody

public Pet getPet(@PathVariable String petId) {

// ...

}

The media type can specify a character set. Negated expressions are supported — e.g. !text/plain means any content type other than "text/plain".

For JSON content type, the UTF-8 charset should be specified even if RFC7159 clearly states that "no charset parameter is defined for this registration" because some browsers require it for interpreting correctly UTF-8 special characters.

You can declare a shared produces attribute at the class level. Unlike most other request mapping attributes however when used at the class level, a method-level produces attribute will overrides rather than extend the class level declaration.

MediaType provides constants for commonly used media types — e.g. APPLICATION\_JSON\_UTF8\_VALUE, APPLICATION\_XML\_VALUE.

Parameters, headers

Same in Spring WebFlux

You can narrow request mappings based on request parameter conditions. You can test for the presence of a request parameter ("myParam"), for the absence ("!myParam"), or for a specific value ("myParam=myValue"):

@GetMapping(path = "/pets/{petId}", params = "myParam=myValue")

public void findPet(@PathVariable String petId) {

// ...

}

You can also use the same with request header conditions:

@GetMapping(path = "/pets", headers = "myHeader=myValue")

public void findPet(@PathVariable String petId) {

// ...

}

You can match Content-Type and Accept with the headers condition but it is better to use consumes and produces instead.

HTTP HEAD, OPTIONS

Same in Spring WebFlux

@GetMapping — and also @RequestMapping(method=HttpMethod.GET), support HTTP HEAD transparently for request mapping purposes. Controller methods don’t need to change. A response wrapper, applied in javax.servlet.http.HttpServlet, ensures a "Content-Length" header is set to the number of bytes written and without actually writing to the response.

@GetMapping — and also @RequestMapping(method=HttpMethod.GET), are implicitly mapped to and also support HTTP HEAD. An HTTP HEAD request is processed as if it were HTTP GET except but instead of writing the body, the number of bytes are counted and the "Content-Length" header set.

By default HTTP OPTIONS is handled by setting the "Allow" response header to the list of HTTP methods listed in all @RequestMapping methods with matching URL patterns.

For a @RequestMapping without HTTP method declarations, the "Allow" header is set to "GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS". Controller methods should always declare the supported HTTP methods for example by using the HTTP method specific variants — @GetMapping, @PostMapping, etc.

@RequestMapping method can be explicitly mapped to HTTP HEAD and HTTP OPTIONS, but that is not necessary in the common case.

Custom Annotations

Same in Spring WebFlux

Spring MVC supports the use of composed annotations for request mapping. Those are annotations that are themselves meta-annotated with @RequestMapping and composed to redeclare a subset (or all) of the @RequestMapping attributes with a narrower, more specific purpose.

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, and @PatchMapping are examples of composed annotations. They’re provided out of the box because arguably most controller methods should be mapped to a specific HTTP method vs using @RequestMapping which by default matches to all HTTP methods. If you need an example of composed annotations, look at how those are declared.

Spring MVC also supports custom request mapping attributes with custom request matching logic. This is a more advanced option that requires sub-classing RequestMappingHandlerMapping and overriding the getCustomMethodCondition method where you can check the custom attribute and return your own RequestCondition.

1.4.3. Handler Methods

Same in Spring WebFlux

@RequestMapping handler methods have a flexible signature and can choose from a range of supported controller method arguments and return values.

Method Arguments

Same in Spring WebFlux

The table below shows supported controller method arguments. Reactive types are not supported for any arguments.

JDK 8’s java.util.Optional is supported as a method argument in combination with annotations that have a required attribute — e.g. @RequestParam, @RequestHeader, etc, and is equivalent to required=false.

Controller method argument Description

WebRequest, NativeWebRequest

Generic access to request parameters, request & session attributes, without direct use of the Servlet API.

javax.servlet.ServletRequest, javax.servlet.ServletResponse

Choose any specific request or response type — e.g. ServletRequest, HttpServletRequest, or Spring’s MultipartRequest, MultipartHttpServletRequest.

javax.servlet.http.HttpSession

Enforces the presence of a session. As a consequence, such an argument is never null.

Note: Session access is not thread-safe. Consider setting the RequestMappingHandlerAdapter's "synchronizeOnSession" flag to "true" if multiple requests are allowed to access a session concurrently.

javax.servlet.http.PushBuilder

Servlet 4.0 push builder API for programmatic HTTP/2 resource pushes. Note that per Servlet spec, the injected PushBuilder instance can be null if the client does not support that HTTP/2 feature.

java.security.Principal

Currently authenticated user; possibly a specific Principal implementation class if known.

HttpMethod

The HTTP method of the request.

java.util.Locale

The current request locale, determined by the most specific LocaleResolver available, in effect, the configured LocaleResolver/LocaleContextResolver.

java.util.TimeZone + java.time.ZoneId

The time zone associated with the current request, as determined by a LocaleContextResolver.

java.io.InputStream, java.io.Reader

For access to the raw request body as exposed by the Servlet API.

java.io.OutputStream, java.io.Writer

For access to the raw response body as exposed by the Servlet API.

@PathVariable

For access to URI template variables. See URI patterns.

@MatrixVariable

For access to name-value pairs in URI path segments. See Matrix variables.

@RequestParam

For access to Servlet request parameters. Parameter values are converted to the declared method argument type. See @RequestParam.

Note that use of @RequestParam is optional, e.g. to set its attributes. See "Any other argument" further below in this table.

@RequestHeader

For access to request headers. Header values are converted to the declared method argument type. See @RequestHeader.

@CookieValue

For access to cookies. Cookies values are converted to the declared method argument type. See @CookieValue.

@RequestBody

For access to the HTTP request body. Body content is converted to the declared method argument type using HttpMessageConverters. See @RequestBody.

HttpEntity<B>

For access to request headers and body. The body is converted with HttpMessageConverters. See HttpEntity.

@RequestPart

For access to a part in a "multipart/form-data" request. See Multipart.

java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap

For access to the model that is used in HTML controllers and exposed to templates as part of view rendering.

RedirectAttributes

Specify attributes to use in case of a redirect — i.e. to be appended to the query string, and/or flash attributes to be stored temporarily until the request after redirect. See Redirect attributes and Flash attributes.

@ModelAttribute

For access to an existing attribute in the model (instantiated if not present) with data binding and validation applied. See @ModelAttribute as well as Model and DataBinder.

Note that use of @ModelAttribute is optional, e.g. to set its attributes. See "Any other argument" further below in this table.

Errors, BindingResult

For access to errors from validation and data binding for a command object (i.e. @ModelAttribute argument), or errors from the validation of an @RequestBody or @RequestPart arguments; an Errors, or BindingResult argument must be declared immediately after the validated method argument.

SessionStatus + class-level @SessionAttributes

For marking form processing complete which triggers cleanup of session attributes declared through a class-level @SessionAttributes annotation. See @SessionAttributes for more details.

UriComponentsBuilder

For preparing a URL relative to the current request’s host, port, scheme, context path, and the literal part of the servlet mapping also taking into account Forwarded and X-Forwarded-\* headers. See URI Links.

@SessionAttribute

For access to any session attribute; in contrast to model attributes stored in the session as a result of a class-level @SessionAttributes declaration. See @SessionAttribute for more details.

@RequestAttribute

For access to request attributes. See @RequestAttribute for more details.

Any other argument

If a method argument is not matched to any of the above, by default it is resolved as an @RequestParam if it is a simple type, as determined by BeanUtils#isSimpleProperty, or as an @ModelAttribute otherwise.

Return Values

Same in Spring WebFlux

The table below shows supported controller method return values. Reactive types are supported for all return values, see below for more details.

Controller method return value Description

@ResponseBody

The return value is converted through HttpMessageConverters and written to the response. See @ResponseBody.

HttpEntity<B>, ResponseEntity<B>

The return value specifies the full response including HTTP headers and body be converted through HttpMessageConverters and written to the response. See ResponseEntity.

HttpHeaders

For returning a response with headers and no body.

String

A view name to be resolved with ViewResolver's and used together with the implicit model — determined through command objects and @ModelAttribute methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).

View

A View instance to use for rendering together with the implicit model — determined through command objects and @ModelAttribute methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).

java.util.Map, org.springframework.ui.Model

Attributes to be added to the implicit model with the view name implicitly determined through a RequestToViewNameTranslator.

@ModelAttribute

An attribute to be added to the model with the view name implicitly determined through a RequestToViewNameTranslator.

Note that @ModelAttribute is optional. See "Any other return value" further below in this table.

ModelAndView object

The view and model attributes to use, and optionally a response status.

void

A method with a void return type (or null return value) is considered to have fully handled the response if it also has a ServletResponse, or an OutputStream argument, or an @ResponseStatus annotation. The same is true also if the controller has made a positive ETag or lastModified timestamp check (see Controllers for details).

If none of the above is true, a void return type may also indicate "no response body" for REST controllers, or default view name selection for HTML controllers.

DeferredResult<V>

Produce any of the above return values asynchronously from any thread — e.g. possibly as a result of some event or callback. See Async Requests and DeferredResult.

Callable<V>

Produce any of the above return values asynchronously in a Spring MVC managed thread. See Async Requests and Callable.

ListenableFuture<V>, java.util.concurrent.CompletionStage<V>, java.util.concurrent.CompletableFuture<V>

Alternative to DeferredResult as a convenience for example when an underlying service returns one of those.

ResponseBodyEmitter, SseEmitter

Emit a stream of objects asynchronously to be written to the response with HttpMessageConverter's; also supported as the body of a ResponseEntity. See Async Requests and HTTP Streaming.

StreamingResponseBody

Write to the response OutputStream asynchronously; also supported as the body of a ResponseEntity. See Async Requests and HTTP Streaming.

Reactive types — Reactor, RxJava, or others via ReactiveAdapterRegistry

Alternative to DeferredResult with multi-value streams (e.g. Flux, Observable) collected to a List.

For streaming scenarios — e.g. text/event-stream, application/json+stream —  SseEmitter and ResponseBodyEmitter are used instead, where ServletOutputStream blocking I/O is performed on a Spring MVC managed thread and back pressure applied against the completion of each write.

See Async Requests and Reactive types.

Any other return value

If a return value is not matched to any of the above, by default it is treated as a view name, if it is String or void (default view name selection via RequestToViewNameTranslator applies); or as a model attribute to be added to the model, unless it is a simple type, as determined by BeanUtils#isSimpleProperty in which case it remains unresolved.

Type Conversion

Same in Spring WebFlux

Some annotated controller method arguments that represent String-based request input — e.g. @RequestParam, @RequestHeader, @PathVariable, @MatrixVariable, and @CookieValue, may require type conversion if the argument is declared as something other than String.

For such cases type conversion is automatically applied based on the configured converters. By default simple types such as int, long, Date, etc. are supported. Type conversion can be customized through a WebDataBinder, see DataBinder, or by registering Formatters with the FormattingConversionService, see Spring Field Formatting.

Matrix variables

Same in Spring WebFlux

RFC 3986 discusses name-value pairs in path segments. In Spring MVC we refer to those as "matrix variables" based on an "old post" by Tim Berners-Lee but they can be also be referred to as URI path parameters.

Matrix variables can appear in any path segment, each variable separated by semicolon and multiple values separated by comma, e.g. "/cars;color=red,green;year=2012". Multiple values can also be specified through repeated variable names, e.g. "color=red;color=green;color=blue".

If a URL is expected to contain matrix variables, the request mapping for a controller method must use a URI variable to mask that variable content and ensure the request can be matched successfully independent of matrix variable order and presence. Below is an example:

// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")

public void findPet(@PathVariable String petId, @MatrixVariable int q) {

// petId == 42

// q == 11

}

Given that all path segments may contain matrix variables, sometimes you may need to disambiguate which path variable the matrix variable is expected to be in. For example:

// GET /owners/42;q=11/pets/21;q=22

@GetMapping("/owners/{ownerId}/pets/{petId}")

public void findPet(

@MatrixVariable(name="q", pathVar="ownerId") int q1,

@MatrixVariable(name="q", pathVar="petId") int q2) {

// q1 == 11

// q2 == 22

}

A matrix variable may be defined as optional and a default value specified:

// GET /pets/42

@GetMapping("/pets/{petId}")

public void findPet(@MatrixVariable(required=false, defaultValue="1") int q) {

// q == 1

}

To get all matrix variables, use a MultiValueMap:

// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@GetMapping("/owners/{ownerId}/pets/{petId}")

public void findPet(

@MatrixVariable MultiValueMap<String, String> matrixVars,

@MatrixVariable(pathVar="petId") MultiValueMap<String, String> petMatrixVars) {

// matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]

// petMatrixVars: ["q" : 22, "s" : 23]

}

Note that you need to enable the use of matrix variables. In the MVC Java config you need to set a UrlPathHelper with removeSemicolonContent=false via Path Matching. In the MVC XML namespace, use <mvc:annotation-driven enable-matrix-variables="true"/>.

@RequestParam

Same in Spring WebFlux

Use the @RequestParam annotation to bind Servlet request parameters (i.e. query parameters or form data) to a method argument in a controller.

The following code snippet shows the usage:

@Controller

@RequestMapping("/pets")

public class EditPetForm {

// ...

@GetMapping

public String setupForm(@RequestParam("petId") int petId, Model model) {

Pet pet = this.clinic.loadPet(petId);

model.addAttribute("pet", pet);

return "petForm";

}

// ...

}

Method parameters using this annotation are required by default, but you can specify that a method parameter is optional by setting @RequestParam's required flag to false or by declaring the argument with an java.util.Optional wrapper.

Type conversion is applied automatically if the target method parameter type is not String. See Type Conversion.

When an @RequestParam annotation is declared as Map<String, String> or MultiValueMap<String, String> argument, the map is populated with all request parameters.

Note that use of @RequestParam is optional, e.g. to set its attributes. By default any argument that is a simple value type, as determined by BeanUtils#isSimpleProperty, and is not resolved by any other argument resolver, is treated as if it was annotated with @RequestParam.

@RequestHeader

Same in Spring WebFlux

Use the @RequestHeader annotation to bind a request header to a method argument in a controller.

Given request with headers:

Host localhost:8080

Accept text/html,application/xhtml+xml,application/xml;q=0.9

Accept-Language fr,en-gb;q=0.7,en;q=0.3

Accept-Encoding gzip,deflate

Accept-Charset ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive 300

The following gets the value of the Accept-Encoding and Keep-Alive headers:

@GetMapping("/demo")

public void handle(

@RequestHeader("Accept-Encoding") String encoding,

@RequestHeader("Keep-Alive") long keepAlive) {

//...

}

Type conversion is applied automatically if the target method parameter type is not String. See Type Conversion.

When an @RequestHeader annotation is used on a Map<String, String>, MultiValueMap<String, String>, or HttpHeaders argument, the map is populated with all header values.

Built-in support is available for converting a comma-separated string into an array/collection of strings or other types known to the type conversion system. For example a method parameter annotated with @RequestHeader("Accept") may be of type String but also String[] or List<String>.

@CookieValue

Same in Spring WebFlux

Use the @CookieValue annotation to bind the value of an HTTP cookie to a method argument in a controller.

Given request with the following cookie:

JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84

The following code sample demonstrates how to get the cookie value:

@GetMapping("/demo")

public void handle(@CookieValue("JSESSIONID") String cookie) {

//...

}

Type conversion is applied automatically if the target method parameter type is not String. See Type Conversion.

@ModelAttribute

Same in Spring WebFlux

Use the @ModelAttribute annotation on a method argument to access an attribute from the model, or have it instantiated if not present. The model attribute is also overlaid with values from HTTP Servlet request parameters whose names match to field names. This is referred to as data binding and it saves you from having to deal with parsing and converting individual query parameters and form fields. For example:

@PostMapping("/owners/{ownerId}/pets/{petId}/edit")

public String processSubmit(@ModelAttribute Pet pet) { }

The Pet instance above is resolved as follows:

From the model if already added via Model.

From the HTTP session via @SessionAttributes.

From a URI path variable passed through a Converter (example below).

From the invocation of a default constructor.

From the invocation of a "primary constructor" with arguments matching to Servlet request parameters; argument names are determined via JavaBeans @ConstructorProperties or via runtime-retained parameter names in the bytecode.

While it is common to use a Model to populate the model with attributes, one other alternative is to rely on a Converter<String, T> in combination with a URI path variable convention. In the example below the model attribute name "account" matches the URI path variable "account" and the Account is loaded by passing the String account number through a registered Converter<String, Account>:

@PutMapping("/accounts/{account}")

public String save(@ModelAttribute("account") Account account) {

// ...

}

After the model attribute instance is obtained, data binding is applied. The WebDataBinder class matches Servlet request parameter names (query parameters and form fields) to field names on the target Object. Matching fields are populated after type conversion is applied where necessary. For more on data binding (and validation) see Validation. For more on customizing data binding see DataBinder.

Data binding may result in errors. By default a BindException is raised but to check for such errors in the controller method, add a BindingResult argument immediately next to the @ModelAttribute as shown below:

@PostMapping("/owners/{ownerId}/pets/{petId}/edit")

public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {

if (result.hasErrors()) {

return "petForm";

}

// ...

}

In some cases you may want access to a model attribute without data binding. For such cases you can inject the Model into the controller and access it directly or alternatively set @ModelAttribute(binding=false) as shown below:

@ModelAttribute

public AccountForm setUpForm() {

return new AccountForm();

}

@ModelAttribute

public Account findAccount(@PathVariable String accountId) {

return accountRepository.findOne(accountId);

}

@PostMapping("update")

public String update(@Valid AccountUpdateForm form, BindingResult result,

@ModelAttribute(binding=false) Account account) {

// ...

}

Validation can be applied automatically after data binding by adding the javax.validation.Valid annotation or Spring’s @Validated annotation (also see Bean validation and Spring validation). For example:

@PostMapping("/owners/{ownerId}/pets/{petId}/edit")

public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult result) {

if (result.hasErrors()) {

return "petForm";

}

// ...

}

Note that use of @ModelAttribute is optional, e.g. to set its attributes. By default any argument that is not a simple value type, as determined by BeanUtils#isSimpleProperty, and is not resolved by any other argument resolver, is treated as if it was annotated with @ModelAttribute.

@SessionAttributes

Same in Spring WebFlux

@SessionAttributes is used to store model attributes in the HTTP Servlet session between requests. It is a type-level annotation that declares session attributes used by a specific controller. This will typically list the names of model attributes or types of model attributes which should be transparently stored in the session for subsequent requests to access.

For example:

@Controller

@SessionAttributes("pet")

public class EditPetForm {

// ...

}

On the first request when a model attribute with the name "pet" is added to the model, it is automatically promoted to and saved in the HTTP Servlet session. It remains there until another controller method uses a SessionStatus method argument to clear the storage:

@Controller

@SessionAttributes("pet")

public class EditPetForm {

// ...

@PostMapping("/pets/{id}")

public String handle(Pet pet, BindingResult errors, SessionStatus status) {

if (errors.hasErrors) {

// ...

}

status.setComplete();

// ...

}

}

}

@SessionAttribute

Same in Spring WebFlux

If you need access to pre-existing session attributes that are managed globally, i.e. outside the controller (e.g. by a filter), and may or may not be present use the @SessionAttribute annotation on a method parameter:

@RequestMapping("/")

public String handle(@SessionAttribute User user) {

// ...

}

For use cases that require adding or removing session attributes consider injecting org.springframework.web.context.request.WebRequest or javax.servlet.http.HttpSession into the controller method.

For temporary storage of model attributes in the session as part of a controller workflow consider using SessionAttributes as described in @SessionAttributes.

@RequestAttribute

Same in Spring WebFlux

Similar to @SessionAttribute the @RequestAttribute annotation can be used to access pre-existing request attributes created earlier, e.g. by a Servlet Filter or HandlerInterceptor:

@GetMapping("/")

public String handle(@RequestAttribute Client client) {

// ...

}

Redirect attributes

By default all model attributes are considered to be exposed as URI template variables in the redirect URL. Of the remaining attributes those that are primitive types or collections/arrays of primitive types are automatically appended as query parameters.

Appending primitive type attributes as query parameters may be the desired result if a model instance was prepared specifically for the redirect. However, in annotated controllers the model may contain additional attributes added for rendering purposes (e.g. drop-down field values). To avoid the possibility of having such attributes appear in the URL, an @RequestMapping method can declare an argument of type RedirectAttributes and use it to specify the exact attributes to make available to RedirectView. If the method does redirect, the content of RedirectAttributes is used. Otherwise the content of the model is used.

The RequestMappingHandlerAdapter provides a flag called "ignoreDefaultModelOnRedirect" that can be used to indicate the content of the default Model should never be used if a controller method redirects. Instead the controller method should declare an attribute of type RedirectAttributes or if it doesn’t do so no attributes should be passed on to RedirectView. Both the MVC namespace and the MVC Java config keep this flag set to false in order to maintain backwards compatibility. However, for new applications we recommend setting it to true

Note that URI template variables from the present request are automatically made available when expanding a redirect URL and do not need to be added explicitly neither through Model nor RedirectAttributes. For example:

@PostMapping("/files/{path}")

public String upload(...) {

// ...

return "redirect:files/{path}";

}

Another way of passing data to the redirect target is via Flash Attributes. Unlike other redirect attributes, flash attributes are saved in the HTTP session (and hence do not appear in the URL). See Flash attributes for more information.

Flash attributes

Flash attributes provide a way for one request to store attributes intended for use in another. This is most commonly needed when redirecting — for example, the Post/Redirect/Get pattern. Flash attributes are saved temporarily before the redirect (typically in the session) to be made available to the request after the redirect and removed immediately.

Spring MVC has two main abstractions in support of flash attributes. FlashMap is used to hold flash attributes while FlashMapManager is used to store, retrieve, and manage FlashMap instances.

Flash attribute support is always "on" and does not need to enabled explicitly although if not used, it never causes HTTP session creation. On each request there is an "input" FlashMap with attributes passed from a previous request (if any) and an "output" FlashMap with attributes to save for a subsequent request. Both FlashMap instances are accessible from anywhere in Spring MVC through static methods in RequestContextUtils.

Annotated controllers typically do not need to work with FlashMap directly. Instead an @RequestMapping method can accept an argument of type RedirectAttributes and use it to add flash attributes for a redirect scenario. Flash attributes added via RedirectAttributes are automatically propagated to the "output" FlashMap. Similarly, after the redirect, attributes from the "input" FlashMap are automatically added to the Model of the controller serving the target URL.

Matching requests to flash attributes

The concept of flash attributes exists in many other Web frameworks and has proven to be exposed sometimes to concurrency issues. This is because by definition flash attributes are to be stored until the next request. However the very "next" request may not be the intended recipient but another asynchronous request (e.g. polling or resource requests) in which case the flash attributes are removed too early.

To reduce the possibility of such issues, RedirectView automatically "stamps" FlashMap instances with the path and query parameters of the target redirect URL. In turn the default FlashMapManager matches that information to incoming requests when looking up the "input" FlashMap.

This does not eliminate the possibility of a concurrency issue entirely but nevertheless reduces it greatly with information that is already available in the redirect URL. Therefore the use of flash attributes is recommended mainly for redirect scenarios .

Multipart

Same in Spring WebFlux

After a MultipartResolver has been enabled, the content of POST requests with "multipart/form-data" is parsed and accessible as regular request parameters. In the example below we access one regular form field and one uploaded file:

@Controller

public class FileUploadController {

@PostMapping("/form")

public String handleFormUpload(@RequestParam("name") String name,

@RequestParam("file") MultipartFile file) {

if (!file.isEmpty()) {

byte[] bytes = file.getBytes();

// store the bytes somewhere

return "redirect:uploadSuccess";

}

return "redirect:uploadFailure";

}

}

When using Servlet 3.0 multipart parsing you can also use javax.servlet.http.Part as a method argument instead of Spring’s MultipartFile.

Multipart content can also be used as part of data binding to a command object. For example the above form field and file could have been fields on a form object:

class MyForm {

private String name;

private MultipartFile file;

// ...

}

@Controller

public class FileUploadController {

@PostMapping("/form")

public String handleFormUpload(MyForm form, BindingResult errors) {

if (!form.getFile().isEmpty()) {

byte[] bytes = form.getFile().getBytes();

// store the bytes somewhere

return "redirect:uploadSuccess";

}

return "redirect:uploadFailure";

}

}

Multipart requests can also be submitted from non-browser clients in a RESTful service scenario. For example a file along with JSON:

POST /someUrl

Content-Type: multipart/mixed

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp

Content-Disposition: form-data; name="meta-data"

Content-Type: application/json; charset=UTF-8

Content-Transfer-Encoding: 8bit

{

"name": "value"

}

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp

Content-Disposition: form-data; name="file-data"; filename="file.properties"

Content-Type: text/xml

Content-Transfer-Encoding: 8bit

... File Data ...

You can access the "meta-data" part with @RequestParam as a String but you’ll probably want it deserialized from JSON (similar to @RequestBody). Use the @RequestPart annotation to access a multipart after converting it with an HttpMessageConverter:

@PostMapping("/")

public String handle(@RequestPart("meta-data") MetaData metadata,

@RequestPart("file-data") MultipartFile file) {

// ...

}

@RequestPart can be used in combination with javax.validation.Valid, or Spring’s @Validated annotation, which causes Standard Bean Validation to be applied. By default validation errors cause a MethodArgumentNotValidException which is turned into a 400 (BAD\_REQUEST) response. Alternatively validation errors can be handled locally within the controller through an Errors or BindingResult argument:

@PostMapping("/")

public String handle(@Valid @RequestPart("meta-data") MetaData metadata,

BindingResult result) {

// ...

}

@RequestBody

Same in Spring WebFlux

Use the @RequestBody annotation to have the request body read and deserialized into an Object through an HttpMessageConverter. Below is an example with an @RequestBody argument:

@PostMapping("/accounts")

public void handle(@RequestBody Account account) {

// ...

}

You can use the Message Converters option of the MVC Config to configure or customize message conversion.

@RequestBody can be used in combination with javax.validation.Valid, or Spring’s @Validated annotation, which causes Standard Bean Validation to be applied. By default validation errors cause a MethodArgumentNotValidException which is turned into a 400 (BAD\_REQUEST) response. Alternatively validation errors can be handled locally within the controller through an Errors or BindingResult argument:

@PostMapping("/accounts")

public void handle(@Valid @RequestBody Account account, BindingResult result) {

// ...

}

HttpEntity

Same in Spring WebFlux

HttpEntity is more or less identical to using @RequestBody but based on a container object that exposes request headers and body. Below is an example:

@PostMapping("/accounts")

public void handle(HttpEntity<Account> entity) {

// ...

}

@ResponseBody

Same in Spring WebFlux

Use the @ResponseBody annotation on a method to have the return serialized to the response body through an HttpMessageConverter. For example:

@GetMapping("/accounts/{id}")

@ResponseBody

public Account handle() {

// ...

}

@ResponseBody is also supported at the class level in which case it is inherited by all controller methods. This is the effect of @RestController which is nothing more than a meta-annotation marked with @Controller and @ResponseBody.

@ResponseBody may be used with reactive types. See Async Requests and Reactive types for more details.

You can use the Message Converters option of the MVC Config to configure or customize message conversion.

@ResponseBody methods can be combined with JSON serialization views. See Jackson JSON for details.

ResponseEntity

Same in Spring WebFlux

ResponseEntity is more or less identical to using @ResponseBody but based on a container object that specifies request headers and body. Below is an example:

@PostMapping("/something")

public ResponseEntity<String> handle() {

// ...

URI location = ... ;

return ResponseEntity.created(location).build();

}

Jackson JSON

Jackson serialization views

Same in Spring WebFlux

Spring MVC provides built-in support for Jackson’s Serialization Views which allows rendering only a subset of all fields in an Object. To use it with @ResponseBody or ResponseEntity controller methods, use Jackson’s @JsonView annotation to activate a serialization view class:

@RestController

public class UserController {

@GetMapping("/user")

@JsonView(User.WithoutPasswordView.class)

public User getUser() {

return new User("eric", "7!jd#h23");

}

}

public class User {

public interface WithoutPasswordView {};

public interface WithPasswordView extends WithoutPasswordView {};

private String username;

private String password;

public User() {

}

public User(String username, String password) {

this.username = username;

this.password = password;

}

@JsonView(WithoutPasswordView.class)

public String getUsername() {

return this.username;

}

@JsonView(WithPasswordView.class)

public String getPassword() {

return this.password;

}

}

@JsonView allows an array of view classes but you can only specify only one per controller method. Use a composite interface if you need to activate multiple views.

For controllers relying on view resolution, simply add the serialization view class to the model:

@Controller

public class UserController extends AbstractController {

@GetMapping("/user")

public String getUser(Model model) {

model.addAttribute("user", new User("eric", "7!jd#h23"));

model.addAttribute(JsonView.class.getName(), User.WithoutPasswordView.class);

return "userView";

}

}

Jackson JSONP

In order to enable JSONP support for @ResponseBody and ResponseEntity methods, declare an @ControllerAdvice bean that extends AbstractJsonpResponseBodyAdvice as shown below where the constructor argument indicates the JSONP query parameter name(s):

@ControllerAdvice

public class JsonpAdvice extends AbstractJsonpResponseBodyAdvice {

public JsonpAdvice() {

super("callback");

}

}

For controllers relying on view resolution, JSONP is automatically enabled when the request has a query parameter named jsonp or callback. Those names can be customized through jsonpParameterNames property.

As of Spring Framework 5.0.7, JSONP support is deprecated and will be removed as of Spring Framework 5.1, CORS should be used instead.

1.4.4. Model

Same in Spring WebFlux

The @ModelAttribute annotation can be used:

On a method argument in @RequestMapping methods to create or access an Object from the model, and to bind it to the request through a WebDataBinder.

As a method-level annotation in @Controller or @ControllerAdvice classes helping to initialize the model prior to any @RequestMapping method invocation.

On a @RequestMapping method to mark its return value is a model attribute.

This section discusses @ModelAttribute methods, or the 2nd from the list above. A controller can have any number of @ModelAttribute methods. All such methods are invoked before @RequestMapping methods in the same controller. A @ModelAttribute method can also be shared across controllers via @ControllerAdvice. See the section on Controller Advice for more details.

@ModelAttribute methods have flexible method signatures. They support many of the same arguments as @RequestMapping methods except for @ModelAttribute itself nor anything related to the request body.

An example @ModelAttribute method:

@ModelAttribute

public void populateModel(@RequestParam String number, Model model) {

model.addAttribute(accountRepository.findAccount(number));

// add more ...

}

To add one attribute only:

@ModelAttribute

public Account addAccount(@RequestParam String number) {

return accountRepository.findAccount(number);

}

When a name is not explicitly specified, a default name is chosen based on the Object type as explained in the Javadoc for Conventions. You can always assign an explicit name by using the overloaded addAttribute method or through the name attribute on @ModelAttribute (for a return value).

@ModelAttribute can also be used as a method-level annotation on @RequestMapping methods in which case the return value of the @RequestMapping method is interpreted as a model attribute. This is typically not required, as it is the default behavior in HTML controllers, unless the return value is a String which would otherwise be interpreted as a view name (also see [mvc-coc-r2vnt]). @ModelAttribute can also help to customize the model attribute name:

@GetMapping("/accounts/{id}")

@ModelAttribute("myAccount")

public Account handle() {

// ...

return account;

}

1.4.5. DataBinder

Same in Spring WebFlux

@Controller or @ControllerAdvice classes can have @InitBinder methods in order to initialize instances of WebDataBinder, and those in turn are used to:

Bind request parameters (i.e. form data or query) to a model object.

Convert String-based request values such as request parameters, path variables, headers, cookies, and others, to the target type of controller method arguments.

Format model object values as String values when rendering HTML forms.

@InitBinder methods can register controller-specific java.bean.PropertyEditor, or Spring Converter and Formatter components. In addition, the MVC config can be used to register Converter and Formatter types in a globally shared FormattingConversionService.

@InitBinder methods support many of the same arguments that a @RequestMapping methods do, except for @ModelAttribute (command object) arguments. Typically they’re are declared with a WebDataBinder argument, for registrations, and a void return value. Below is an example:

@Controller

public class FormController {

@InitBinder

public void initBinder(WebDataBinder binder) {

SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

dateFormat.setLenient(false);

binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));

}

// ...

}

Alternatively when using a Formatter-based setup through a shared FormattingConversionService, you could re-use the same approach and register controller-specific Formatter's:

@Controller

public class FormController {

@InitBinder

protected void initBinder(WebDataBinder binder) {

binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));

}

// ...

}

1.4.6. Exceptions

Same in Spring WebFlux

@Controller and @ControllerAdvice classes can have @ExceptionHandler methods to handle exceptions from controller methods. For example:

@Controller

public class SimpleController {

// ...

@ExceptionHandler

public ResponseEntity<String> handle(IOException ex) {

// ...

}

}

The exception may match against a top-level exception being propagated (i.e. a direct IOException thrown), or against the immediate cause within a top-level wrapper exception (e.g. an IOException wrapped inside an IllegalStateException).

For matching exception types, preferably declare the target exception as a method argument as shown above. When multiple exception methods match, a root exception match is generally preferred to a cause exception match. More specifically, the ExceptionDepthComparator is used to sort exceptions based on their depth from the thrown exception type.

Alternatively, the annotation declaration may narrow the exception types to match:

@ExceptionHandler({FileSystemException.class, RemoteException.class})

public ResponseEntity<String> handle(IOException ex) {

// ...

}

Or even a list of specific exception types with a very generic argument signature:

@ExceptionHandler({FileSystemException.class, RemoteException.class})

public ResponseEntity<String> handle(Exception ex) {

// ...

}

The distinction between root and cause exception matching can be surprising:

In the IOException variant above, the method will typically be called with the actual FileSystemException or RemoteException instance as the argument since both of them extend from IOException. However, if any such matching exception is propagated within a wrapper exception which is an IOException itself, the passed-in exception instance will be that wrapper exception.

The behavior is even simpler in the handle(Exception) variant: This will always be invoked with the wrapper exception in a wrapping scenario, with the actually matching exception to be found through ex.getCause() in that case. The passed-in exception will only be the actual FileSystemException or RemoteException instance when these are thrown as top-level exceptions.

We generally recommend to be as specific as possible in the argument signature, reducing the potential for mismatches between root and cause exception types. Consider breaking a multi-matching method into individual @ExceptionHandler methods, each matching a single specific exception type through its signature.

In a multi-@ControllerAdvice arrangement, please declare your primary root exception mappings on a @ControllerAdvice prioritized with a corresponding order. While a root exception match is preferred to a cause, this is defined among the methods of a given controller or @ControllerAdvice class. This means a cause match on a higher-priority @ControllerAdvice bean is preferred to any match (e.g. root) on a lower-priority @ControllerAdvice bean.

Last but not least, an @ExceptionHandler method implementation may choose to back out of dealing with a given exception instance by rethrowing it in its original form. This is useful in scenarios where you are only interested in root-level matches or in matches within a specific context that cannot be statically determined. A rethrown exception will be propagated through the remaining resolution chain, just like if the given @ExceptionHandler method would not have matched in the first place.

Support for @ExceptionHandler methods in Spring MVC is built on the DispatcherServlet level, HandlerExceptionResolver mechanism.

Method arguments

@ExceptionHandler methods support the following arguments:

Method argument Description

Exception type

For access to the raised exception.

HandlerMethod

For access to the controller method that raised the exception.

WebRequest, NativeWebRequest

Generic access to request parameters, request & session attributes, without direct use of the Servlet API.

javax.servlet.ServletRequest, javax.servlet.ServletResponse

Choose any specific request or response type — e.g. ServletRequest, HttpServletRequest, or Spring’s MultipartRequest, MultipartHttpServletRequest.

javax.servlet.http.HttpSession

Enforces the presence of a session. As a consequence, such an argument is never null.

Note: Session access is not thread-safe. Consider setting the RequestMappingHandlerAdapter's "synchronizeOnSession" flag to "true" if multiple requests are allowed to access a session concurrently.

java.security.Principal

Currently authenticated user; possibly a specific Principal implementation class if known.

HttpMethod

The HTTP method of the request.

java.util.Locale

The current request locale, determined by the most specific LocaleResolver available, in effect, the configured LocaleResolver/LocaleContextResolver.

java.util.TimeZone + java.time.ZoneId

The time zone associated with the current request, as determined by a LocaleContextResolver.

java.io.OutputStream, java.io.Writer

For access to the raw response body as exposed by the Servlet API.

java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap

For access to the model for an error response, always empty.

RedirectAttributes

Specify attributes to use in case of a redirect — i.e. to be appended to the query string, and/or flash attributes to be stored temporarily until the request after redirect. See Redirect attributes and Flash attributes.

@SessionAttribute

For access to any session attribute; in contrast to model attributes stored in the session as a result of a class-level @SessionAttributes declaration. See @SessionAttribute for more details.

@RequestAttribute

For access to request attributes. See @RequestAttribute for more details.

Return Values

@ExceptionHandler methods support the following return values:

Return value Description

@ResponseBody

The return value is converted through HttpMessageConverters and written to the response. See @ResponseBody.

HttpEntity<B>, ResponseEntity<B>

The return value specifies the full response including HTTP headers and body be converted through HttpMessageConverters and written to the response. See ResponseEntity.

String

A view name to be resolved with ViewResolver's and used together with the implicit model — determined through command objects and @ModelAttribute methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).

View

A View instance to use for rendering together with the implicit model — determined through command objects and @ModelAttribute methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).

java.util.Map, org.springframework.ui.Model

Attributes to be added to the implicit model with the view name implicitly determined through a RequestToViewNameTranslator.

@ModelAttribute

An attribute to be added to the model with the view name implicitly determined through a RequestToViewNameTranslator.

Note that @ModelAttribute is optional. See "Any other return value" further below in this table.

ModelAndView object

The view and model attributes to use, and optionally a response status.

void

A method with a void return type (or null return value) is considered to have fully handled the response if it also has a ServletResponse, or an OutputStream argument, or an @ResponseStatus annotation. The same is true also if the controller has made a positive ETag or lastModified timestamp check (see Controllers for details).

If none of the above is true, a void return type may also indicate "no response body" for REST controllers, or default view name selection for HTML controllers.

Any other return value

If a return value is not matched to any of the above, by default it is treated as a model attribute to be added to the model, unless it is a simple type, as determined by BeanUtils#isSimpleProperty in which case it remains unresolved.

REST API exceptions

Same in Spring WebFlux

A common requirement for REST services is to include error details in the body of the response. The Spring Framework does not automatically do this because the representation of error details in the response body is application specific. However a @RestController may use @ExceptionHandler methods with a ResponseEntity return value to set the status and the body of the response. Such methods may also be declared in @ControllerAdvice classes to apply them globally.

Applications that implement global exception handling with error details in the response body should consider extending ResponseEntityExceptionHandler which provides handling for exceptions that Spring MVC raises along with hooks to customize the response body. To make use of this, create a subclass of ResponseEntityExceptionHandler, annotate with @ControllerAdvice, override the necessary methods, and declare it as a Spring bean.

1.4.7. Controller Advice

Same in Spring WebFlux

Typically @ExceptionHandler, @InitBinder, and @ModelAttribute methods apply within the @Controller class (or class hierarchy) they are declared in. If you want such methods to apply more globally, across controllers, you can declare them in a class marked with @ControllerAdvice or @RestControllerAdvice.

@ControllerAdvice is marked with @Component which means such classes can be registered as Spring beans via component scanning. @RestControllerAdvice is also a meta-annotation marked with both @ControllerAdvice and @ResponseBody which essentially means @ExceptionHandler methods are rendered to the response body via message conversion (vs view resolution/template rendering).

On startup, the infrastructure classes for @RequestMapping and @ExceptionHandler methods detect Spring beans of type @ControllerAdvice, and then apply their methods at runtime. Global @ExceptionHandler methods (from an @ControllerAdvice) are applied after local ones (from the @Controller). By contrast global @ModelAttribute and @InitBinder methods are applied before local ones.

By default @ControllerAdvice methods apply to every request, i.e. all controllers, but you can narrow that down to a subset of controllers via attributes on the annotation:

// Target all Controllers annotated with @RestController

@ControllerAdvice(annotations = RestController.class)

public class ExampleAdvice1 {}

// Target all Controllers within specific packages

@ControllerAdvice("org.example.controllers")

public class ExampleAdvice2 {}

// Target all Controllers assignable to specific classes

@ControllerAdvice(assignableTypes = {ControllerInterface.class, AbstractController.class})

public class ExampleAdvice3 {}

Keep in mind the above selectors are evaluated at runtime and may negatively impact performance if used extensively. See the @ControllerAdvice Javadoc for more details.

1.5. URI Links

Same in Spring WebFlux

This section describes various options available in the Spring Framework to work with URI’s.

1.5.1. UriComponents

Spring MVC and Spring WebFlux

UriComponentsBuilder helps to build URI’s from URI templates with variables:

UriComponents uriComponents = UriComponentsBuilder

.fromUriString("http://example.com/hotels/{hotel}")

.queryParam("q", "{q}")

.encode()

.build();

URI uri = uriComponents.expand("Westin", "123").toUri();

Static factory method with a URI template.

Add and/or replace URI components.

Request to have the URI template and URI variables encoded.

Build a UriComponents.

Expand variables, and obtain the URI.

The above can be consolidated into one chain and shortened with buildAndExpand:

URI uri = UriComponentsBuilder

.fromUriString("http://example.com/hotels/{hotel}")

.queryParam("q", "{q}")

.encode()

.buildAndExpand("Westin", "123")

.toUri();

It can be shortened further by going directly to URI (which implies encoding):

URI uri = UriComponentsBuilder

.fromUriString("http://example.com/hotels/{hotel}")

.queryParam("q", "{q}")

.build("Westin", "123");

Or shorter further yet, with a full URI template:

URI uri = UriComponentsBuilder

.fromUriString("http://example.com/hotels/{hotel}?q={q}")

.build("Westin", "123");

1.5.2. UriBuilder

Spring MVC and Spring WebFlux

UriComponentsBuilder implements UriBuilder. A UriBuilder in turn can be created with a UriBuilderFactory. Together UriBuilderFactory and UriBuilder provide a pluggable mechanism to build URIs from URI templates, based on shared configuration such as a base url, encoding preferences, and others.

The RestTemplate and the WebClient can be configured with a UriBuilderFactory to customize the preparation of URIs. DefaultUriBuilderFactory is a default implementation of UriBuilderFactory that uses UriComponentsBuilder internally and exposes shared configuration options.

RestTemplate example:

// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "http://example.org";

DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);

factory.setEncodingMode(EncodingMode.TEMPLATE\_AND\_VARIABLES);

RestTemplate restTemplate = new RestTemplate();

restTemplate.setUriTemplateHandler(factory);

WebClient example:

// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "http://example.org";

DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);

factory.setEncodingMode(EncodingMode.TEMPLATE\_AND\_VARIABLES);

WebClient client = WebClient.builder().uriBuilderFactory(factory).build();

In addition DefaultUriBuilderFactory can also be used directly. It is similar to using UriComponentsBuilder but instead of static factory methods, it is an actual instance that holds configuration and preferences:

String baseUrl = "http://example.com";

DefaultUriBuilderFactory uriBuilderFactory = new DefaultUriBuilderFactory(baseUrl);

URI uri = uriBuilderFactory.uriString("/hotels/{hotel}")

.queryParam("q", "{q}")

.build("Westin", "123");

1.5.3. URI Encoding

Spring MVC and Spring WebFlux

UriComponentsBuilder exposes encoding options at 2 levels:

UriComponentsBuilder#encode() - pre-encodes the URI template first, then strictly encodes URI variables when expanded.

UriComponents#encode() - encodes URI components after URI variables are expanded.

Both options replace non-ASCII and illegal characters with escaped octets, however option 1 also replaces characters with reserved meaning that appear in URI variables.

Consider ";" which is legal in a path but has reserved meaning. Option 1 replaces ";" with "%3B" in URI variables but not in the URI template. By contrast, option 2 never replaces ";" since it is a legal character in a path.

For most cases option 1 is likely to give the expected result because it treats URI variables as opaque data to be fully encoded, while option 2 is useful only if URI variables intentionally contain reserved characters.

Example usage using option 1:

URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")

.queryParam("q", "{q}")

.encode()

.buildAndExpand("New York", "foo+bar")

.toUri();

// Result is "/hotel%20list/New%20York?foo%2Bbar"

The above can be shortened by going directly to URI (which implies encoding):

URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")

.queryParam("q", "{q}")

.build("New York", "foo+bar")

Or shorter further yet, with a full URI template:

URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}?q={q}")

.build("New York", "foo+bar")

The WebClient and the RestTemplate expand and encode URI templates internally through the UriBuilderFactory strategy. Both can be configured with a custom strategy:

String baseUrl = "http://example.com";

DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl)

factory.setEncodingMode(EncodingMode.TEMPLATE\_AND\_VALUES);

// Customize the RestTemplate..

RestTemplate restTemplate = new RestTemplate();

restTemplate.setUriTemplateHandler(factory);

// Customize the WebClient..

WebClient client = WebClient.builder().uriBuilderFactory(factory).build();

The DefaultUriBuilderFactory implementation uses UriComponentsBuilder internally to expand and encode URI templates. As a factory it provides a single place to configure the approach to encoding based on one of the below encoding modes:

TEMPLATE\_AND\_VALUES — uses UriComponentsBuilder#encode(), corresponding to option 1 above, to pre-encode the URI template and strictly encode URI variables when expanded.

VALUES\_ONLY — does not encode the URI template and instead applies strict encoding to URI variables via UriUtils#encodeUriUriVariables prior to expanding them into the template.

URI\_COMPONENTS — uses UriComponents#encode(), corresponding to option 2 above, to encode URI component value after URI variables are expanded.

NONE — no encoding is applied.

Out of the box the RestTemplate is set to EncodingMode.URI\_COMPONENTS for historic reasons and for backwards compatibility. The WebClient relies on the default value in DefaultUriBuilderFactory which was changed from EncodingMode.URI\_COMPONENTS in 5.0.x to EncodingMode.TEMPLATE\_AND\_VALUES in 5.1.

1.5.4. Servlet request relative

You can use ServletUriComponentsBuilder to create URIs relative to the current request:

HttpServletRequest request = ...

// Re-uses host, scheme, port, path and query string...

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromRequest(request)

.replaceQueryParam("accountId", "{id}").build()

.expand("123")

.encode();

You can create URIs relative to the context path:

// Re-uses host, port and context path...

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromContextPath(request)

.path("/accounts").build()

You can create URIs relative to a Servlet (e.g. /main/\*):

// Re-uses host, port, context path, and Servlet prefix...

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromServletMapping(request)

.path("/accounts").build()

ServletUriComponentsBuilder detects and uses information from the "Forwarded", "X-Forwarded-Host", "X-Forwarded-Port", and "X-Forwarded-Proto" headers, so the resulting links reflect the original request. You need to ensure that your application is behind a trusted proxy which filters out such headers coming from outside. Also consider using the ForwardedHeaderFilter which processes such headers once per request, and also provides an option to remove and ignore such headers.

1.5.5. Links to controllers

Spring MVC provides a mechanism to prepare links to controller methods. For example, the following MVC controller easily allows for link creation:

@Controller

@RequestMapping("/hotels/{hotel}")

public class BookingController {

@GetMapping("/bookings/{booking}")

public ModelAndView getBooking(@PathVariable Long booking) {

// ...

}

}

You can prepare a link by referring to the method by name:

UriComponents uriComponents = MvcUriComponentsBuilder

.fromMethodName(BookingController.class, "getBooking", 21).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();

In the above example we provided actual method argument values, in this case the long value 21, to be used as a path variable and inserted into the URL. Furthermore, we provided the value 42 in order to fill in any remaining URI variables such as the "hotel" variable inherited from the type-level request mapping. If the method had more arguments you can supply null for arguments not needed for the URL. In general only @PathVariable and @RequestParam arguments are relevant for constructing the URL.

There are additional ways to use MvcUriComponentsBuilder. For example you can use a technique akin to mock testing through proxies to avoid referring to the controller method by name (the example assumes static import of MvcUriComponentsBuilder.on):

UriComponents uriComponents = MvcUriComponentsBuilder

.fromMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();

Controller method signatures are limited in their design when supposed to be usable for link creation with fromMethodCall. Aside from needing a proper parameter signature, there is a technical limitation on the return type: namely generating a runtime proxy for link builder invocations, so the return type must not be final. In particular, the common String return type for view names does not work here; use ModelAndView or even plain Object (with a String return value) instead.

The above examples use static methods in MvcUriComponentsBuilder. Internally they rely on ServletUriComponentsBuilder to prepare a base URL from the scheme, host, port, context path and servlet path of the current request. This works well in most cases, however sometimes it may be insufficient. For example you may be outside the context of a request (e.g. a batch process that prepares links) or perhaps you need to insert a path prefix (e.g. a locale prefix that was removed from the request path and needs to be re-inserted into links).

For such cases you can use the static "fromXxx" overloaded methods that accept a UriComponentsBuilder to use base URL. Or you can create an instance of MvcUriComponentsBuilder with a base URL and then use the instance-based "withXxx" methods. For example:

UriComponentsBuilder base = ServletUriComponentsBuilder.fromCurrentContextPath().path("/en");

MvcUriComponentsBuilder builder = MvcUriComponentsBuilder.relativeTo(base);

builder.withMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();

MvcUriComponentsBuilder detects and uses information from the "Forwarded", "X-Forwarded-Host", "X-Forwarded-Port", and "X-Forwarded-Proto" headers, so the resulting links reflect the original request. You need to ensure that your application is behind a trusted proxy which filters out such headers coming from outside. Also consider using the ForwardedHeaderFilter which processes such headers once per request, and also provides an option to remove and ignore such headers.

1.5.6. Links in views

You can also build links to annotated controllers from views such as JSP, Thymeleaf, FreeMarker. This can be done using the fromMappingName method in MvcUriComponentsBuilder which refers to mappings by name.

Every @RequestMapping is assigned a default name based on the capital letters of the class and the full method name. For example, the method getFoo in class FooController is assigned the name "FC#getFoo". This strategy can be replaced or customized by creating an instance of HandlerMethodMappingNamingStrategy and plugging it into your RequestMappingHandlerMapping. The default strategy implementation also looks at the name attribute on @RequestMapping and uses that if present. That means if the default mapping name assigned conflicts with another (e.g. overloaded methods) you can assign a name explicitly on the @RequestMapping.

The assigned request mapping names are logged at TRACE level on startup.

The Spring JSP tag library provides a function called mvcUrl that can be used to prepare links to controller methods based on this mechanism.

For example given:

@RequestMapping("/people/{id}/addresses")

public class PersonAddressController {

@RequestMapping("/{country}")

public HttpEntity getAddress(@PathVariable String country) { ... }

}

You can prepare a link from a JSP as follows:

<%@ taglib uri="http://www.springframework.org/tags" prefix="s" %>

...

<a href="${s:mvcUrl('PAC#getAddress').arg(0,'US').buildAndExpand('123')}">Get Address</a>

The above example relies on the mvcUrl JSP function declared in the Spring tag library (i.e. META-INF/spring.tld). For more advanced cases (e.g. a custom base URL as explained in the previous section), it is easy to define your own function, or use a custom tag file, in order to use a specific instance of MvcUriComponentsBuilder with a custom base URL.

1.6. Async Requests

Compared to WebFlux

Spring MVC has an extensive integration with Servlet 3.0 asynchronous request processing:

DeferredResult and Callable return values in controller method provide basic support for a single asynchronous return value.

Controllers can stream multiple values including SSE and raw data.

Controllers can use reactive clients and return reactive types for response handling.

1.6.1. DeferredResult

Compared to WebFlux

Once the asynchronous request processing feature is enabled in the Servlet container, controller methods can wrap any supported controller method return value with DeferredResult:

@GetMapping("/quotes")

@ResponseBody

public DeferredResult<String> quotes() {

DeferredResult<String> deferredResult = new DeferredResult<String>();

// Save the deferredResult somewhere..

return deferredResult;

}

// From some other thread...

deferredResult.setResult(data);

The controller can produce the return value asynchronously, from a different thread, for example in response to an external event (JMS message), a scheduled task, or other.

1.6.2. Callable

Compared to WebFlux

A controller may also wrap any supported return value with java.util.concurrent.Callable:

@PostMapping

public Callable<String> processUpload(final MultipartFile file) {

return new Callable<String>() {

public String call() throws Exception {

// ...

return "someView";

}

};

}

The return value will then be obtained by executing the the given task through the configured TaskExecutor.

1.6.3. Processing

Compared to WebFlux

Here is a very concise overview of Servlet asynchronous request processing:

A ServletRequest can be put in asynchronous mode by calling request.startAsync(). The main effect of doing so is that the Servlet, as well as any Filters, can exit but the response will remain open to allow processing to complete later.

The call to request.startAsync() returns AsyncContext which can be used for further control over async processing. For example it provides the method dispatch, that is similar to a forward from the Servlet API except it allows an application to resume request processing on a Servlet container thread.

The ServletRequest provides access to the current DispatcherType that can be used to distinguish between processing the initial request, an async dispatch, a forward, and other dispatcher types.

DeferredResult processing:

Controller returns a DeferredResult and saves it in some in-memory queue or list where it can be accessed.

Spring MVC calls request.startAsync().

Meanwhile the DispatcherServlet and all configured Filter’s exit the request processing thread but the response remains open.

The application sets the DeferredResult from some thread and Spring MVC dispatches the request back to the Servlet container.

The DispatcherServlet is invoked again and processing resumes with the asynchronously produced return value.

Callable processing:

Controller returns a Callable.

Spring MVC calls request.startAsync() and submits the Callable to a TaskExecutor for processing in a separate thread.

Meanwhile the DispatcherServlet and all Filter’s exit the Servlet container thread but the response remains open.

Eventually the Callable produces a result and Spring MVC dispatches the request back to the Servlet container to complete processing.

The DispatcherServlet is invoked again and processing resumes with the asynchronously produced return value from the Callable.

For further background and context you can also read the blog posts that introduced asynchronous request processing support in Spring MVC 3.2.

Exception handling

When using a DeferredResult you can choose whether to call setResult or setErrorResult with an exception. In both cases Spring MVC dispatches the request back to the Servlet container to complete processing. It is then treated either as if the controller method returned the given value, or as if it produced the given exception. The exception then goes through the regular exception handling mechanism, e.g. invoking @ExceptionHandler methods.

When using Callable, similar processing logic follows. The main difference being that the result is returned from the Callable or an exception is raised by it.

Interception

HandlerInterceptor's can also be AsyncHandlerInterceptor in order to receive the afterConcurrentHandlingStarted callback on the initial request that starts asynchronous processing instead of postHandle and afterCompletion.

HandlerInterceptor's can also register a CallableProcessingInterceptor or a DeferredResultProcessingInterceptor in order to integrate more deeply with the lifecycle of an asynchronous request for example to handle a timeout event. See AsyncHandlerInterceptor for more details.

DeferredResult provides onTimeout(Runnable) and onCompletion(Runnable) callbacks. See the Javadoc of DeferredResult for more details. Callable can be substituted for WebAsyncTask that exposes additional methods for timeout and completion callbacks.

Compared to WebFlux

The Servlet API was originally built for making a single pass through the Filter-Servlet chain. Asynchronous request processing, added in Servlet 3.0, allows applications to exit the Filter-Servlet chain but leave the response open for further processing. The Spring MVC async support is built around that mechanism. When a controller returns a DeferredResult, the Filter-Servlet chain is exited and the Servlet container thread is released. Later when the DeferredResult is set, an ASYNC dispatch (to the same URL) is made during which the controller is mapped again but rather than invoking it, the DeferredResult value is used (as if the controller returned it) to resume processing.

By contrast Spring WebFlux is neither built on the Servlet API, nor does it need such an asynchronous request processing feature because it is asynchronous by design. Asynchronous handling is built into all framework contracts and is intrinsically supported through :: stages of request processing.

From a programming model perspective, both Spring MVC and Spring WebFlux support asynchronous and Reactive types as return values in controller methods. Spring MVC even supports streaming, including reactive back pressure. However individual writes to the response remain blocking (and performed on a separate thread) unlike WebFlux that relies on non-blocking I/O and does not need an extra thread for each write.

Another fundamental difference is that Spring MVC does not support asynchronous or reactive types in controller method arguments, e.g. @RequestBody, @RequestPart, and others, nor does it have any explicit support for asynchronous and reactive types as model attributes. Spring WebFlux does support all that.

1.6.4. HTTP Streaming

Same in Spring WebFlux

DeferredResult and Callable can be used for a single asynchronous return value. What if you want to produce multiple asynchronous values and have those written to the response?

Objects

The ResponseBodyEmitter return value can be used to produce a stream of Objects, where each Object sent is serialized with an HttpMessageConverter and written to the response. For example:

@GetMapping("/events")

public ResponseBodyEmitter handle() {

ResponseBodyEmitter emitter = new ResponseBodyEmitter();

// Save the emitter somewhere..

return emitter;

}

// In some other thread

emitter.send("Hello once");

// and again later on

emitter.send("Hello again");

// and done at some point

emitter.complete();

ResponseBodyEmitter can also be used as the body in a ResponseEntity allowing you to customize the status and headers of the response.

When an emitter throws an IOException (e.g. if the remote client went away) applications are not responsible for cleaning up the connection, and should not invoke emitter.complete or emitter.completeWithError. Instead the servlet container automatically initiates an AsyncListener error notification in which Spring MVC makes a completeWithError call, which in turn performs one a final ASYNC dispatch to the application during which Spring MVC invokes the configured exception resolvers and completes the request.

SSE

SseEmitter is a sub-class of ResponseBodyEmitter that provides support for Server-Sent Events where events sent from the server are formatted according to the W3C SSE specification. In order to produce an SSE stream from a controller simply return SseEmitter:

@GetMapping(path="/events", produces=MediaType.TEXT\_EVENT\_STREAM\_VALUE)

public SseEmitter handle() {

SseEmitter emitter = new SseEmitter();

// Save the emitter somewhere..

return emitter;

}

// In some other thread

emitter.send("Hello once");

// and again later on

emitter.send("Hello again");

// and done at some point

emitter.complete();

While SSE is the main option for streaming into browsers, note that Internet Explorer does not support Server-Sent Events. Consider using Spring’s WebSocket messaging with SockJS fallback transports (including SSE) that target a wide range of browsers.

Also see previous section for notes on exception handling.

Raw data

Sometimes it is useful to bypass message conversion and stream directly to the response OutputStream for example for a file download. Use the of the StreamingResponseBody return value type to do that:

@GetMapping("/download")

public StreamingResponseBody handle() {

return new StreamingResponseBody() {

@Override

public void writeTo(OutputStream outputStream) throws IOException {

// write...

}

};

}

StreamingResponseBody can be used as the body in a ResponseEntity allowing you to customize the status and headers of the response.

1.6.5. Reactive types

Same in Spring WebFlux

Spring MVC supports use of reactive client libraries in a controller. This includes the WebClient from spring-webflux and others such as Spring Data reactive data repositories. In such scenarios it is convenient to be able to return reactive types from the controller method .

Reactive return values are handled as follows:

A single-value promise is adapted to, and similar to using DeferredResult. Examples include Mono (Reactor) or Single (RxJava).

A multi-value stream, with a streaming media type such as "application/stream+json" or "text/event-stream", is adapted to, and similar to using ResponseBodyEmitter or SseEmitter. Examples include Flux (Reactor) or Observable (RxJava). Applications can also return Flux<ServerSentEvent> or Observable<ServerSentEvent>.

A multi-value stream, with any other media type (e.g. "application/json"), is adapted to, and similar to using DeferredResult<List<?>>.

Spring MVC supports Reactor and RxJava through the ReactiveAdapterRegistry from spring-core which allows it to adapt from multiple reactive libraries.

For streaming to the response, reactive back pressure is supported, but writes to the response are still blocking, and are executed on a separate thread through the configured TaskExecutor in order to avoid blocking the upstream source (e.g. a Flux returned from the WebClient). By default SimpleAsyncTaskExecutor is used for the blocking writes but that is not suitable under load. If you plan to stream with a reactive type, please use the MVC config to configure a task executor.

1.6.6. Disconnects

Same in Spring WebFlux

The Servlet API does not provide any notification when a remote client goes away. Therefore while streaming to the response, whether via SseEmitter or <<mvc-ann-async-reactive-types,reactive types>, it is important to send data periodically, since the write would fail if the client has disconnected. The send could take the form of an empty (comment-only) SSE event, or any other data that the other side would have to to interpret as a heartbeat and ignore.

Alternatively consider using web messaging solutions such as STOMP over WebSocket or WebSocket with SockJS that have a built-in heartbeat mechanism.

1.6.7. Configuration

Compared to WebFlux

The async request processing feature must be enabled at the Servlet container level. The MVC config also exposes several options for asynchronous requests.

Servlet container

Filter and Servlet declarations have an asyncSupported that needs to be set to true in order enable asynchronous request processing. In addition, Filter mappings should be declared to handle the ASYNC javax.servlet.DispatchType.

In Java configuration, when you use AbstractAnnotationConfigDispatcherServletInitializer to initialize the Servlet container, this is done automatically.

In web.xml configuration, add <async-supported>true</async-supported> to the DispatcherServlet and to Filter declarations, and also add <dispatcher>ASYNC</dispatcher> to filter mappings.

Spring MVC

The MVC config exposes options related to async request processing:

Java config — use the configureAsyncSupport callback on WebMvcConfigurer.

XML namespace — use the <async-support> element under <mvc:annotation-driven>.

You can configure the following:

Default timeout value for async requests, which if not set, depends on the underlying Servlet container (e.g. 10 seconds on Tomcat).

AsyncTaskExecutor to use for blocking writes when streaming with Reactive types, and also for executing Callable's returned from controller methods. It is highly recommended to configure this property if you’re streaming with reactive types or have controller methods that return Callable since by default it is a SimpleAsyncTaskExecutor.

DeferredResultProcessingInterceptor's and CallableProcessingInterceptor's.

Note that the default timeout value can also be set on a DeferredResult, ResponseBodyEmitter and SseEmitter. For a Callable, use WebAsyncTask to provide a timeout value.

1.7. CORS

Same in Spring WebFlux

1.7.1. Introduction

Same in Spring WebFlux

For security reasons browsers prohibit AJAX calls to resources outside the current origin. For example you could have your bank account in one tab and evil.com in another. Scripts from evil.com should not be able to make AJAX requests to your bank API with your credentials, e.g. withdrawing money from your account!

Cross-Origin Resource Sharing (CORS) is a W3C specification implemented by most browsers that allows you to specify what kind of cross domain requests are authorized rather than using less secure and less powerful workarounds based on IFRAME or JSONP.

1.7.2. Processing

Same in Spring WebFlux

The CORS specification distinguishes between preflight, simple, and actual requests. To learn how CORS works, you can read this article, among many others, or refer to the specification for more details.

Spring MVC HandlerMapping's provide built-in support for CORS. After successfully mapping a request to a handler, HandlerMapping's check the CORS configuration for the given request and handler and take further actions. Preflight requests are handled directly while simple and actual CORS requests are intercepted, validated, and have required CORS response headers set.

In order to enable cross-origin requests (i.e. the Origin header is present and differs from the host of the request) you need to have some explicitly declared CORS configuration. If no matching CORS configuration is found, preflight requests are rejected. No CORS headers are added to the responses of simple and actual CORS requests and consequently browsers reject them.

Each HandlerMapping can be configured individually with URL pattern based CorsConfiguration mappings. In most cases applications will use the MVC Java config or the XML namespace to declare such mappings, which results in a single, global map passed to all HadlerMappping's.

Global CORS configuration at the HandlerMapping level can be combined with more fine-grained, handler-level CORS configuration. For example annotated controllers can use class or method-level @CrossOrigin annotations (other handlers can implement CorsConfigurationSource).

The rules for combining global and local configuration are generally additive — e.g. all global and all local origins. For those attributes where only a single value can be accepted such as allowCredentials and maxAge, the local overrides the global value. See CorsConfiguration#combine(CorsConfiguration) for more details.

To learn more from the source or make advanced customizations, check:

CorsConfiguration

CorsProcessor, DefaultCorsProcessor

AbstractHandlerMapping

1.7.3. @CrossOrigin

Same in Spring WebFlux

The @CrossOrigin annotation enables cross-origin requests on annotated controller methods:

@RestController

@RequestMapping("/account")

public class AccountController {

@CrossOrigin

@GetMapping("/{id}")

public Account retrieve(@PathVariable Long id) {

// ...

}

@DeleteMapping("/{id}")

public void remove(@PathVariable Long id) {

// ...

}

}

By default @CrossOrigin allows:

All origins.

All headers.

All HTTP methods to which the controller method is mapped.

allowedCredentials is not enabled by default since that establishes a trust level that exposes sensitive user-specific information such as cookies and CSRF tokens, and should only be used where appropriate.

maxAge is set to 30 minutes.

@CrossOrigin is supported at the class level too and inherited by all methods:

@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)

@RestController

@RequestMapping("/account")

public class AccountController {

@GetMapping("/{id}")

public Account retrieve(@PathVariable Long id) {

// ...

}

@DeleteMapping("/{id}")

public void remove(@PathVariable Long id) {

// ...

}

}

CrossOrigin can be used at both class and method-level:

@CrossOrigin(maxAge = 3600)

@RestController

@RequestMapping("/account")

public class AccountController {

@CrossOrigin("http://domain2.com")

@GetMapping("/{id}")

public Account retrieve(@PathVariable Long id) {

// ...

}

@DeleteMapping("/{id}")

public void remove(@PathVariable Long id) {

// ...

}

}

1.7.4. Global Config

Same in Spring WebFlux

In addition to fine-grained, controller method level configuration you’ll probably want to define some global CORS configuration too. You can set URL-based CorsConfiguration mappings individually on any HandlerMapping. Most applications however will use the MVC Java config or the MVC XNM namespace to do that.

By default global configuration enables the following:

All origins.

All headers.

GET, HEAD, and POST methods.

allowedCredentials is not enabled by default since that establishes a trust level that exposes sensitive user-specific information such as cookies and CSRF tokens, and should only be used where appropriate.

maxAge is set to 30 minutes.

Java Config

Same in Spring WebFlux

To enable CORS in the MVC Java config, use the CorsRegistry callback:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/api/\*\*")

.allowedOrigins("http://domain2.com")

.allowedMethods("PUT", "DELETE")

.allowedHeaders("header1", "header2", "header3")

.exposedHeaders("header1", "header2")

.allowCredentials(true).maxAge(3600);

// Add more mappings...

}

}

XML Config

To enable CORS in the XML namespace, use the <mvc:cors> element:

<mvc:cors>

<mvc:mapping path="/api/\*\*"

allowed-origins="http://domain1.com, http://domain2.com"

allowed-methods="GET, PUT"

allowed-headers="header1, header2, header3"

exposed-headers="header1, header2" allow-credentials="true"

max-age="123" />

<mvc:mapping path="/resources/\*\*"

allowed-origins="http://domain1.com" />

</mvc:cors>

1.7.5. CORS Filter

Same in Spring WebFlux

You can apply CORS support through the built-in CorsFilter.

If you’re trying to use the CorsFilter with Spring Security, keep in mind that Spring Security has built-in support for CORS.

To configure the filter pass a CorsConfigurationSource to its constructor:

CorsConfiguration config = new CorsConfiguration();

// Possibly...

// config.applyPermitDefaultValues()

config.setAllowCredentials(true);

config.addAllowedOrigin("http://domain1.com");

config.addAllowedHeader("");

config.addAllowedMethod("");

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

source.registerCorsConfiguration("/\*\*", config);

CorsFilter filter = new CorsFilter(source);

1.8. Web Security

Same in Spring WebFlux

The Spring Security project provides support for protecting web applications from malicious exploits. Check out the Spring Security reference documentation including:

Spring MVC Security

Spring MVC Test Support

CSRF protection

Security Response Headers

HDIV is another web security framework that integrates with Spring MVC.

1.9. HTTP Caching

Same in Spring WebFlux

HTTP caching can significantly improve the performance of a web application. HTTP caching revolves around the "Cache-Control" response header and subsequently conditional request headers such as "Last-Modified" and "ETag". "Cache-Control" advises private (e.g. browser) and public (e.g. proxy) caches how to cache and re-use responses. An "ETag" header is used to make a conditional request that may result in a 304 (NOT\_MODIFIED) without a body, if the content has not changed. "ETag" can be seen as a more sophisticated successor to the Last-Modified header.

This section describes HTTP caching related options available in Spring Web MVC.

1.9.1. CacheControl

Same in Spring WebFlux

CacheControl provides support for configuring settings related to the "Cache-Control" header and is accepted as an argument in a number of places:

WebContentInterceptor

WebContentGenerator

Controllers

Static resources

While RFC 7234 describes all possible directives for the "Cache-Control" response header, the CacheControl type takes a use case oriented approach focusing on the common scenarios:

// Cache for an hour - "Cache-Control: max-age=3600"

CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

// Prevent caching - "Cache-Control: no-store"

CacheControl ccNoStore = CacheControl.noStore();

// Cache for ten days in public and private caches,

// public caches should not transform the response

// "Cache-Control: max-age=864000, public, no-transform"

CacheControl ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS).noTransform().cachePublic();

WebContentGenerator also accept a simpler cachePeriod property, in seconds, that works as follows:

A -1 value won’t generate a "Cache-Control" response header.

A 0 value will prevent caching using the 'Cache-Control: no-store' directive.

An n > 0 value will cache the given response for n seconds using the 'Cache-Control: max-age=n' directive.

1.9.2. Controllers

Same in Spring WebFlux

Controllers can add explicit support for HTTP caching. This is recommended since the lastModified or ETag value for a resource needs to be calculated before it can be compared against conditional request headers. A controller can add an ETag and "Cache-Control" settings to a ResponseEntity:

@GetMapping("/book/{id}")

public ResponseEntity<Book> showBook(@PathVariable Long id) {

Book book = findBook(id);

String version = book.getVersion();

return ResponseEntity

.ok()

.cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))

.eTag(version) // lastModified is also available

.body(book);

}

This will send an 304 (NOT\_MODIFIED) response with an empty body, if the comparison to the conditional request headers indicates the content has not changed. Otherwise the "ETag" and "Cache-Control" headers will be added to the response.

The check against conditional request headers can also be made in the controller:

@RequestMapping

public String myHandleMethod(WebRequest webRequest, Model model) {

long eTag = ...

if (request.checkNotModified(eTag)) {

return null;

}

model.addAttribute(...);

return "myViewName";

}

Application-specific calculation.

Response has been set to 304 (NOT\_MODIFIED), no further processing.

Continue with request processing.

There are 3 variants for checking conditional requests against eTag values, lastModified values, or both. For conditional "GET" and "HEAD" requests, the response may be set to 304 (NOT\_MODIFIED). For conditional "POST", "PUT", and "DELETE", the response would be set to 409 (PRECONDITION\_FAILED) instead to prevent concurrent modification.

1.9.3. Static resources

Same in Spring WebFlux

Static resources should be served with a "Cache-Control" and conditional response headers for optimal performance. See section on configuring Static Resources.

1.9.4. ETag Filter

The ShallowEtagHeaderFilter can be used to add "shallow" eTag values, computed from the response content and thus saving bandwith but not CPU time. See Shallow ETag.

1.10. View Technologies

Same in Spring WebFlux

The use of view technologies in Spring MVC is pluggable, whether you decide to use Thymeleaf, Groovy Markup Templates, JSPs, or other, is primarily a matter of a configuration change. This chapter covers view technologies integrated with Spring MVC. We assume you are already familiar with View Resolution.

1.10.1. Thymeleaf

Same in Spring WebFlux

Thymeleaf is modern server-side Java template engine that emphasizes natural HTML templates that can be previewed in a browser by double-clicking, which is very helpful for independent work on UI templates, e.g. by designer, without the need for a running server. If you’re looking to replace JSPs, Thymeleaf offers one of the most extensive set of features that will make such a transition easier. Thymeleaf is actively developed and maintained. For a more complete introduction see the Thymeleaf project home page.

The Thymeleaf integration with Spring MVC is managed by the Thymeleaf project. The configuration involves a few bean declarations such as ServletContextTemplateResolver, SpringTemplateEngine, and ThymeleafViewResolver. See Thymeleaf+Spring for more details.

1.10.2. FreeMarker

Same in Spring WebFlux

Apache FreeMarker is a template engine for generating any kind of text output from HTML to email, and others. The Spring Framework has a built-in integration for using Spring MVC with FreeMarker templates.

View config

Same in Spring WebFlux

To configure FreeMarker as a view technology:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.freemarker();

}

// Configure FreeMarker...

@Bean

public FreeMarkerConfigurer freeMarkerConfigurer() {

FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();

configurer.setTemplateLoaderPath("/WEB-INF/freemarker");

return configurer;

}

}

To configure the same in XML:

<mvc:annotation-driven/>

<mvc:view-resolvers>

<mvc:freemarker/>

</mvc:view-resolvers>

<!-- Configure FreeMarker... -->

<mvc:freemarker-configurer>

<mvc:template-loader-path location="/WEB-INF/freemarker"/>

</mvc:freemarker-configurer>

Or you can also declare the FreeMarkerConfigurer bean for full control over all properties:

<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">

<property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>

</bean>

Your templates need to be stored in the directory specified by the FreeMarkerConfigurer shown above. Given the above configuration if your controller returns the view name "welcome" then the resolver will look for the /WEB-INF/freemarker/welcome.ftl template.

FreeMarker config

Same in Spring WebFlux

FreeMarker 'Settings' and 'SharedVariables' can be passed directly to the FreeMarker Configuration object managed by Spring by setting the appropriate bean properties on the FreeMarkerConfigurer bean. The freemarkerSettings property requires a java.util.Properties object and the freemarkerVariables property requires a java.util.Map.

<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">

<property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>

<property name="freemarkerVariables">

<map>

<entry key="xml\_escape" value-ref="fmXmlEscape"/>

</map>

</property>

</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>

See the FreeMarker documentation for details of settings and variables as they apply to the Configuration object.

Form handling

Spring provides a tag library for use in JSP’s that contains, amongst others, a <spring:bind/> tag. This tag primarily enables forms to display values from form backing objects and to show the results of failed validations from a Validator in the web or business tier. Spring also has support for the same functionality in FreeMarker, with additional convenience macros for generating form input elements themselves.

The bind macros

A standard set of macros are maintained within the spring-webmvc.jar file for both languages, so they are always available to a suitably configured application.

Some of the macros defined in the Spring libraries are considered internal (private) but no such scoping exists in the macro definitions making all macros visible to calling code and user templates. The following sections concentrate only on the macros you need to be directly calling from within your templates. If you wish to view the macro code directly, the file is called spring.ftl in the package org.springframework.web.servlet.view.freemarker.

Simple binding

In your HTML forms (vm / ftl templates) which act as a form view for a Spring MVC controller, you can use code similar to the following to bind to field values and display error messages for each input field in similar fashion to the JSP equivalent. Example code is shown below for the personForm view configured earlier:

<!-- freemarker macros have to be imported into a namespace. We strongly

recommend sticking to 'spring' -->

<#import "/spring.ftl" as spring/>

<html>

...

<form action="" method="POST">

Name:

<@spring.bind "myModelObject.name"/>

<input type="text"

name="${spring.status.expression}"

value="${spring.status.value?html}"/><br>

<#list spring.status.errorMessages as error> <b>${error}</b> <br> </#list>

<br>

...

<input type="submit" value="submit"/>

</form>

...

</html>

<@spring.bind> requires a 'path' argument which consists of the name of your command object (it will be 'command' unless you changed it in your FormController properties) followed by a period and the name of the field on the command object you wish to bind to. Nested fields can be used too such as "command.address.street". The bind macro assumes the default HTML escaping behavior specified by the ServletContext parameter defaultHtmlEscape in web.xml.

The optional form of the macro called <@spring.bindEscaped> takes a second argument and explicitly specifies whether HTML escaping should be used in the status error messages or values. Set to true or false as required. Additional form handling macros simplify the use of HTML escaping and these macros should be used wherever possible. They are explained in the next section.

Input macros

Additional convenience macros for both languages simplify both binding and form generation (including validation error display). It is never necessary to use these macros to generate form input fields, and they can be mixed and matched with simple HTML or calls direct to the spring bind macros highlighted previously.

The following table of available macros show the FTL definitions and the parameter list that each takes.

Table 6. Table of macro definitions

macro FTL definition

message (output a string from a resource bundle based on the code parameter)

<@spring.message code/>

messageText (output a string from a resource bundle based on the code parameter, falling back to the value of the default parameter)

<@spring.messageText code, text/>

url (prefix a relative URL with the application’s context root)

<@spring.url relativeUrl/>

formInput (standard input field for gathering user input)

<@spring.formInput path, attributes, fieldType/>

formHiddenInput \* (hidden input field for submitting non-user input)

<@spring.formHiddenInput path, attributes/>

formPasswordInput \* (standard input field for gathering passwords. Note that no value will ever be populated in fields of this type)

<@spring.formPasswordInput path, attributes/>

formTextarea (large text field for gathering long, freeform text input)

<@spring.formTextarea path, attributes/>

formSingleSelect (drop down box of options allowing a single required value to be selected)

<@spring.formSingleSelect path, options, attributes/>

formMultiSelect (a list box of options allowing the user to select 0 or more values)

<@spring.formMultiSelect path, options, attributes/>

formRadioButtons (a set of radio buttons allowing a single selection to be made from the available choices)

<@spring.formRadioButtons path, options separator, attributes/>

formCheckboxes (a set of checkboxes allowing 0 or more values to be selected)

<@spring.formCheckboxes path, options, separator, attributes/>

formCheckbox (a single checkbox)

<@spring.formCheckbox path, attributes/>

showErrors (simplify display of validation errors for the bound field)

<@spring.showErrors separator, classOrStyle/>

In FTL (FreeMarker), formHiddenInput and formPasswordInput are not actually required as you can use the normal formInput macro, specifying hidden or password as the value for the fieldType parameter.

The parameters to any of the above macros have consistent meanings:

path: the name of the field to bind to (ie "command.name")

options: a Map of all the available values that can be selected from in the input field. The keys to the map represent the values that will be POSTed back from the form and bound to the command object. Map objects stored against the keys are the labels displayed on the form to the user and may be different from the corresponding values posted back by the form. Usually such a map is supplied as reference data by the controller. Any Map implementation can be used depending on required behavior. For strictly sorted maps, a SortedMap such as a TreeMap with a suitable Comparator may be used and for arbitrary Maps that should return values in insertion order, use a LinkedHashMap or a LinkedMap from commons-collections.

separator: where multiple options are available as discreet elements (radio buttons or checkboxes), the sequence of characters used to separate each one in the list (ie "<br>").

attributes: an additional string of arbitrary tags or text to be included within the HTML tag itself. This string is echoed literally by the macro. For example, in a textarea field you may supply attributes as 'rows="5" cols="60"' or you could pass style information such as 'style="border:1px solid silver"'.

classOrStyle: for the showErrors macro, the name of the CSS class that the span tag wrapping each error will use. If no information is supplied (or the value is empty) then the errors will be wrapped in <b></b> tags.

Examples of the macros are outlined below some in FTL and some in VTL. Where usage differences exist between the two languages, they are explained in the notes.

Input Fields

The formInput macro takes the path parameter (command.name) and an additional attributes parameter which is empty in the example above. The macro, along with all other form generation macros, performs an implicit spring bind on the path parameter. The binding remains valid until a new bind occurs so the showErrors macro doesn’t need to pass the path parameter again - it simply operates on whichever field a bind was last created for.

The showErrors macro takes a separator parameter (the characters that will be used to separate multiple errors on a given field) and also accepts a second parameter, this time a class name or style attribute. Note that FreeMarker is able to specify default values for the attributes parameter.

<@spring.formInput "command.name"/>

<@spring.showErrors "<br>"/>

Output is shown below of the form fragment generating the name field, and displaying a validation error after the form was submitted with no value in the field. Validation occurs through Spring’s Validation framework.

The generated HTML looks like this:

Name:

<input type="text" name="name" value="">

<br>

<b>required</b>

<br>

<br>

The formTextarea macro works the same way as the formInput macro and accepts the same parameter list. Commonly, the second parameter (attributes) will be used to pass style information or rows and cols attributes for the textarea.

Selection Fields

Four selection field macros can be used to generate common UI value selection inputs in your HTML forms.

formSingleSelect

formMultiSelect

formRadioButtons

formCheckboxes

Each of the four macros accepts a Map of options containing the value for the form field, and the label corresponding to that value. The value and the label can be the same.

An example of radio buttons in FTL is below. The form backing object specifies a default value of 'London' for this field and so no validation is necessary. When the form is rendered, the entire list of cities to choose from is supplied as reference data in the model under the name 'cityMap'.

...

Town:

<@spring.formRadioButtons "command.address.town", cityMap, ""/><br><br>

This renders a line of radio buttons, one for each value in cityMap using the separator "". No additional attributes are supplied (the last parameter to the macro is missing). The cityMap uses the same String for each key-value pair in the map. The map’s keys are what the form actually submits as POSTed request parameters, map values are the labels that the user sees. In the example above, given a list of three well known cities and a default value in the form backing object, the HTML would be

Town:

<input type="radio" name="address.town" value="London">London</input>

<input type="radio" name="address.town" value="Paris" checked="checked">Paris</input>

<input type="radio" name="address.town" value="New York">New York</input>

If your application expects to handle cities by internal codes for example, the map of codes would be created with suitable keys like the example below.

protected Map<String, String> referenceData(HttpServletRequest request) throws Exception {

Map<String, String> cityMap = new LinkedHashMap<>();

cityMap.put("LDN", "London");

cityMap.put("PRS", "Paris");

cityMap.put("NYC", "New York");

Map<String, String> model = new HashMap<>();

model.put("cityMap", cityMap);

return model;

}

The code would now produce output where the radio values are the relevant codes but the user still sees the more user friendly city names.

Town:

<input type="radio" name="address.town" value="LDN">London</input>

<input type="radio" name="address.town" value="PRS" checked="checked">Paris</input>

<input type="radio" name="address.town" value="NYC">New York</input>

HTML escaping

Default usage of the form macros above will result in HTML tags that are HTML 4.01 compliant and that use the default value for HTML escaping defined in your web.xml as used by Spring’s bind support. In order to make the tags XHTML compliant or to override the default HTML escaping value, you can specify two variables in your template (or in your model where they will be visible to your templates). The advantage of specifying them in the templates is that they can be changed to different values later in the template processing to provide different behavior for different fields in your form.

To switch to XHTML compliance for your tags, specify a value of true for a model/context variable named xhtmlCompliant:

<#-- for FreeMarker -->

<#assign xhtmlCompliant = true>

Any tags generated by the Spring macros will now be XHTML compliant after processing this directive.

In similar fashion, HTML escaping can be specified per field:

<#-- until this point, default HTML escaping is used -->

<#assign htmlEscape = true>

<#-- next field will use HTML escaping -->

<@spring.formInput "command.name"/>

<#assign htmlEscape = false in spring>

<#-- all future fields will be bound with HTML escaping off -->

1.10.3. Groovy Markup

Groovy Markup Template Engine is primarily aimed at generating XML-like markup (XML, XHTML, HTML5, etc) but that can be used to generate any text based content. The Spring Framework has a built-in integration for using Spring MVC with Groovy Markup.

The Groovy Markup Tempalte engine requires Groovy 2.3.1+.

Configuration

To configure the Groovy Markup Template Engine:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.groovy();

}

// Configure the Groovy Markup Template Engine...

@Bean

public GroovyMarkupConfigurer groovyMarkupConfigurer() {

GroovyMarkupConfigurer configurer = new GroovyMarkupConfigurer();

configurer.setResourceLoaderPath("/WEB-INF/");

return configurer;

}

}

To configure the same in XML:

<mvc:annotation-driven/>

<mvc:view-resolvers>

<mvc:groovy/>

</mvc:view-resolvers>

<!-- Configure the Groovy Markup Template Engine... -->

<mvc:groovy-configurer resource-loader-path="/WEB-INF/"/>

Example

Unlike traditional template engines, Groovy Markup relies on a DSL that uses a builder syntax. Here is a sample template for an HTML page:

yieldUnescaped '<!DOCTYPE html>'

html(lang:'en') {

head {

meta('http-equiv':'"Content-Type" content="text/html; charset=utf-8"')

title('My page')

}

body {

p('This is an example of HTML contents')

}

}

1.10.4. Script Views

Same in Spring WebFlux

The Spring Framework has a built-in integration for using Spring MVC with any templating library that can run on top of the JSR-223 Java scripting engine. Below is a list of templating libraries we’ve tested on different script engines:

Handlebars

Nashorn

Mustache

Nashorn

React

Nashorn

EJS

Nashorn

ERB

JRuby

String templates

Jython

Kotlin Script templating

Kotlin

The basic rule for integrating any other script engine is that it must implement the ScriptEngine and Invocable interfaces.

Requirements

Same in Spring WebFlux

You need to have the script engine on your classpath:

Nashorn JavaScript engine is provided with Java 8+. Using the latest update release available is highly recommended.

JRuby should be added as a dependency for Ruby support.

Jython should be added as a dependency for Python support.

org.jetbrains.kotlin:kotlin-script-util dependency and a META-INF/services/javax.script.ScriptEngineFactory file containing a org.jetbrains.kotlin.script.jsr223.KotlinJsr223JvmLocalScriptEngineFactory line should be added for Kotlin script support, see this example for more details.

You need to have the script templating library. One way to do that for Javascript is through WebJars.

Script templates

Same in Spring WebFlux

Declare a ScriptTemplateConfigurer bean in order to specify the script engine to use, the script files to load, what function to call to render templates, and so on. Below is an example with Mustache templates and the Nashorn JavaScript engine:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.scriptTemplate();

}

@Bean

public ScriptTemplateConfigurer configurer() {

ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();

configurer.setEngineName("nashorn");

configurer.setScripts("mustache.js");

configurer.setRenderObject("Mustache");

configurer.setRenderFunction("render");

return configurer;

}

}

The same in XML:

<mvc:annotation-driven/>

<mvc:view-resolvers>

<mvc:script-template/>

</mvc:view-resolvers>

<mvc:script-template-configurer engine-name="nashorn" render-object="Mustache" render-function="render">

<mvc:script location="mustache.js"/>

</mvc:script-template-configurer>

The controller would look no different:

@Controller

public class SampleController {

@GetMapping("/sample")

public String test(Model model) {

model.addObject("title", "Sample title");

model.addObject("body", "Sample body");

return "template";

}

}

And the Mustache template is:

<html>

<head>

<title>{{title}}</title>

</head>

<body>

<p>{{body}}</p>

</body>

</html>

The render function is called with the following parameters:

String template: the template content

Map model: the view model

RenderingContext renderingContext: the RenderingContext that gives access to the application context, the locale, the template loader and the url (since 5.0)

Mustache.render() is natively compatible with this signature, so you can call it directly.

If your templating technology requires some customization, you may provide a script that implements a custom render function. For example, Handlerbars needs to compile templates before using them, and requires a polyfill in order to emulate some browser facilities not available in the server-side script engine.

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.scriptTemplate();

}

@Bean

public ScriptTemplateConfigurer configurer() {

ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();

configurer.setEngineName("nashorn");

configurer.setScripts("polyfill.js", "handlebars.js", "render.js");

configurer.setRenderFunction("render");

configurer.setSharedEngine(false);

return configurer;

}

}

Setting the sharedEngine property to false is required when using non thread-safe script engines with templating libraries not designed for concurrency, like Handlebars or React running on Nashorn for example. In that case, Java 8u60 or greater is required due to this bug.

polyfill.js only defines the window object needed by Handlebars to run properly:

var window = {};

This basic render.js implementation compiles the template before using it. A production ready implementation should also store and reused cached templates / pre-compiled templates. This can be done on the script side, as well as any customization you need (managing template engine configuration for example).

function render(template, model) {

var compiledTemplate = Handlebars.compile(template);

return compiledTemplate(model);

}

Check out the Spring Framework unit tests, java, and resources, for more configuration examples.

1.10.5. JSP & JSTL

The Spring Framework has a built-in integration for using Spring MVC with JSP and JSTL.

View resolvers

When developing with JSPs you can declare a InternalResourceViewResolver or a ResourceBundleViewResolver bean.

ResourceBundleViewResolver relies on a properties file to define the view names mapped to a class and a URL. With a ResourceBundleViewResolver you can mix different types of views using only one resolver. Here is an example:

<!-- the ResourceBundleViewResolver -->

<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

<property name="basename" value="views"/>

</bean>

# And a sample properties file is uses (views.properties in WEB-INF/classes):

welcome.(class)=org.springframework.web.servlet.view.JstlView

welcome.url=/WEB-INF/jsp/welcome.jsp

productList.(class)=org.springframework.web.servlet.view.JstlView

productList.url=/WEB-INF/jsp/productlist.jsp

InternalResourceBundleViewResolver can also be used for JSPs. As a best practice, we strongly encourage placing your JSP files in a directory under the 'WEB-INF' directory so there can be no direct access by clients.

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>

<property name="prefix" value="/WEB-INF/jsp/"/>

<property name="suffix" value=".jsp"/>

</bean>

JSPs versus JSTL

When using the Java Standard Tag Library you must use a special view class, the JstlView, as JSTL needs some preparation before things such as the I18N features will work.

Spring’s JSP tag library

Spring provides data binding of request parameters to command objects as described in earlier chapters. To facilitate the development of JSP pages in combination with those data binding features, Spring provides a few tags that make things even easier. All Spring tags haveHTML escaping features to enable or disable escaping of characters.

The spring.tld tag library descriptor (TLD) is included in the spring-webmvc.jar. For a comprehensive reference on individual tags, browse the API reference or see the tag library description.

Spring’s form tag library

As of version 2.0, Spring provides a comprehensive set of data binding-aware tags for handling form elements when using JSP and Spring Web MVC. Each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and intuitive to use. The tag-generated HTML is HTML 4.01/XHTML 1.0 compliant.

Unlike other form/input tag libraries, Spring’s form tag library is integrated with Spring Web MVC, giving the tags access to the command object and reference data your controller deals with. As you will see in the following examples, the form tags make JSPs easier to develop, read and maintain.

Let’s go through the form tags and look at an example of how each tag is used. We have included generated HTML snippets where certain tags require further commentary.

Configuration

The form tag library comes bundled in spring-webmvc.jar. The library descriptor is called spring-form.tld.

To use the tags from this library, add the following directive to the top of your JSP page:

<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

where form is the tag name prefix you want to use for the tags from this library.

The form tag

This tag renders an HTML 'form' tag and exposes a binding path to inner tags for binding. It puts the command object in the PageContext so that the command object can be accessed by inner tags. All the other tags in this library are nested tags of the form tag.

Let’s assume we have a domain object called User. It is a JavaBean with properties such as firstName and lastName. We will use it as the form backing object of our form controller which returns form.jsp. Below is an example of what form.jsp would look like:

<form:form>

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName"/></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName"/></td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form:form>

The firstName and lastName values are retrieved from the command object placed in the PageContext by the page controller. Keep reading to see more complex examples of how inner tags are used with the form tag.

The generated HTML looks like a standard form:

<form method="POST">

<table>

<tr>

<td>First Name:</td>

<td><input name="firstName" type="text" value="Harry"/></td>

</tr>

<tr>

<td>Last Name:</td>

<td><input name="lastName" type="text" value="Potter"/></td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form>

The preceding JSP assumes that the variable name of the form backing object is 'command'. If you have put the form backing object into the model under another name (definitely a best practice), then you can bind the form to the named variable like so:

<form:form modelAttribute="user">

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName"/></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName"/></td>

</tr>

<tr>

<td colspan="2">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form:form>

The input tag

This tag renders an HTML 'input' tag using the bound value and type='text' by default. For an example of this tag, see The form tag. Starting with Spring 3.1 you can use other types such HTML5-specific types like 'email', 'tel', 'date', and others.

The checkbox tag

This tag renders an HTML 'input' tag with type 'checkbox'.

Let’s assume our User has preferences such as newsletter subscription and a list of hobbies. Below is an example of the Preferences class:

public class Preferences {

private boolean receiveNewsletter;

private String[] interests;

private String favouriteWord;

public boolean isReceiveNewsletter() {

return receiveNewsletter;

}

public void setReceiveNewsletter(boolean receiveNewsletter) {

this.receiveNewsletter = receiveNewsletter;

}

public String[] getInterests() {

return interests;

}

public void setInterests(String[] interests) {

this.interests = interests;

}

public String getFavouriteWord() {

return favouriteWord;

}

public void setFavouriteWord(String favouriteWord) {

this.favouriteWord = favouriteWord;

}

}

The form.jsp would look like:

<form:form>

<table>

<tr>

<td>Subscribe to newsletter?:</td>

<%-- Approach 1: Property is of type java.lang.Boolean --%>

<td><form:checkbox path="preferences.receiveNewsletter"/></td>

</tr>

<tr>

<td>Interests:</td>

<%-- Approach 2: Property is of an array or of type java.util.Collection --%>

<td>

Quidditch: <form:checkbox path="preferences.interests" value="Quidditch"/>

Herbology: <form:checkbox path="preferences.interests" value="Herbology"/>

Defence Against the Dark Arts: <form:checkbox path="preferences.interests" value="Defence Against the Dark Arts"/>

</td>

</tr>

<tr>

<td>Favourite Word:</td>

<%-- Approach 3: Property is of type java.lang.Object --%>

<td>

Magic: <form:checkbox path="preferences.favouriteWord" value="Magic"/>

</td>

</tr>

</table>

</form:form>

There are 3 approaches to the checkbox tag which should meet all your checkbox needs.

Approach One - When the bound value is of type java.lang.Boolean, the input(checkbox) is marked as 'checked' if the bound value is true. The value attribute corresponds to the resolved value of the setValue(Object) value property.

Approach Two - When the bound value is of type array or java.util.Collection, the input(checkbox) is marked as 'checked' if the configured setValue(Object) value is present in the bound Collection.

Approach Three - For any other bound value type, the input(checkbox) is marked as 'checked' if the configured setValue(Object) is equal to the bound value.

Note that regardless of the approach, the same HTML structure is generated. Below is an HTML snippet of some checkboxes:

<tr>

<td>Interests:</td>

<td>

Quidditch: <input name="preferences.interests" type="checkbox" value="Quidditch"/>

<input type="hidden" value="1" name="\_preferences.interests"/>

Herbology: <input name="preferences.interests" type="checkbox" value="Herbology"/>

<input type="hidden" value="1" name="\_preferences.interests"/>

Defence Against the Dark Arts: <input name="preferences.interests" type="checkbox" value="Defence Against the Dark Arts"/>

<input type="hidden" value="1" name="\_preferences.interests"/>

</td>

</tr>

What you might not expect to see is the additional hidden field after each checkbox. When a checkbox in an HTML page is not checked, its value will not be sent to the server as part of the HTTP request parameters once the form is submitted, so we need a workaround for this quirk in HTML in order for Spring form data binding to work. The checkbox tag follows the existing Spring convention of including a hidden parameter prefixed by an underscore ("\_") for each checkbox. By doing this, you are effectively telling Spring that "the checkbox was visible in the form and I want my object to which the form data will be bound to reflect the state of the checkbox no matter what".

The checkboxes tag

This tag renders multiple HTML 'input' tags with type 'checkbox'.

Building on the example from the previous checkbox tag section. Sometimes you prefer not to have to list all the possible hobbies in your JSP page. You would rather provide a list at runtime of the available options and pass that in to the tag. That is the purpose of the checkboxes tag. You pass in an Array, a List or a Map containing the available options in the "items" property. Typically the bound property is a collection so it can hold multiple values selected by the user. Below is an example of the JSP using this tag:

<form:form>

<table>

<tr>

<td>Interests:</td>

<td>

<%-- Property is of an array or of type java.util.Collection --%>

<form:checkboxes path="preferences.interests" items="${interestList}"/>

</td>

</tr>

</table>

</form:form>

This example assumes that the "interestList" is a List available as a model attribute containing strings of the values to be selected from. In the case where you use a Map, the map entry key will be used as the value and the map entry’s value will be used as the label to be displayed. You can also use a custom object where you can provide the property names for the value using "itemValue" and the label using "itemLabel".

The radiobutton tag

This tag renders an HTML 'input' tag with type 'radio'.

A typical usage pattern will involve multiple tag instances bound to the same property but with different values.

<tr>

<td>Sex:</td>

<td>

Male: <form:radiobutton path="sex" value="M"/> <br/>

Female: <form:radiobutton path="sex" value="F"/>

</td>

</tr>

The radiobuttons tag

This tag renders multiple HTML 'input' tags with type 'radio'.

Just like the checkboxes tag above, you might want to pass in the available options as a runtime variable. For this usage you would use the radiobuttons tag. You pass in an Array, a List or a Map containing the available options in the "items" property. In the case where you use a Map, the map entry key will be used as the value and the map entry’s value will be used as the label to be displayed. You can also use a custom object where you can provide the property names for the value using "itemValue" and the label using "itemLabel".

<tr>

<td>Sex:</td>

<td><form:radiobuttons path="sex" items="${sexOptions}"/></td>

</tr>

The password tag

This tag renders an HTML 'input' tag with type 'password' using the bound value.

<tr>

<td>Password:</td>

<td>

<form:password path="password"/>

</td>

</tr>

Please note that by default, the password value is not shown. If you do want the password value to be shown, then set the value of the 'showPassword' attribute to true, like so.

<tr>

<td>Password:</td>

<td>

<form:password path="password" value="^76525bvHGq" showPassword="true"/>

</td>

</tr>

The select tag

This tag renders an HTML 'select' element. It supports data binding to the selected option as well as the use of nested option and options tags.

Let’s assume a User has a list of skills.

<tr>

<td>Skills:</td>

<td><form:select path="skills" items="${skills}"/></td>

</tr>

If the User’s skill were in Herbology, the HTML source of the 'Skills' row would look like:

<tr>

<td>Skills:</td>

<td>

<select name="skills" multiple="true">

<option value="Potions">Potions</option>

<option value="Herbology" selected="selected">Herbology</option>

<option value="Quidditch">Quidditch</option>

</select>

</td>

</tr>

The option tag

This tag renders an HTML 'option'. It sets 'selected' as appropriate based on the bound value.

<tr>

<td>House:</td>

<td>

<form:select path="house">

<form:option value="Gryffindor"/>

<form:option value="Hufflepuff"/>

<form:option value="Ravenclaw"/>

<form:option value="Slytherin"/>

</form:select>

</td>

</tr>

If the User’s house was in Gryffindor, the HTML source of the 'House' row would look like:

<tr>

<td>House:</td>

<td>

<select name="house">

<option value="Gryffindor" selected="selected">Gryffindor</option>

<option value="Hufflepuff">Hufflepuff</option>

<option value="Ravenclaw">Ravenclaw</option>

<option value="Slytherin">Slytherin</option>

</select>

</td>

</tr>

The options tag

This tag renders a list of HTML 'option' tags. It sets the 'selected' attribute as appropriate based on the bound value.

<tr>

<td>Country:</td>

<td>

<form:select path="country">

<form:option value="-" label="--Please Select"/>

<form:options items="${countryList}" itemValue="code" itemLabel="name"/>

</form:select>

</td>

</tr>

If the User lived in the UK, the HTML source of the 'Country' row would look like:

<tr>

<td>Country:</td>

<td>

<select name="country">

<option value="-">--Please Select</option>

<option value="AT">Austria</option>

<option value="UK" selected="selected">United Kingdom</option>

<option value="US">United States</option>

</select>

</td>

</tr>

As the example shows, the combined usage of an option tag with the options tag generates the same standard HTML, but allows you to explicitly specify a value in the JSP that is for display only (where it belongs) such as the default string in the example: "-- Please Select".

The items attribute is typically populated with a collection or array of item objects. itemValue and itemLabel simply refer to bean properties of those item objects, if specified; otherwise, the item objects themselves will be stringified. Alternatively, you may specify a Map of items, in which case the map keys are interpreted as option values and the map values correspond to option labels. If itemValue and/or itemLabel happen to be specified as well, the item value property will apply to the map key and the item label property will apply to the map value.

The textarea tag

This tag renders an HTML 'textarea'.

<tr>

<td>Notes:</td>

<td><form:textarea path="notes" rows="3" cols="20"/></td>

<td><form:errors path="notes"/></td>

</tr>

The hidden tag

This tag renders an HTML 'input' tag with type 'hidden' using the bound value. To submit an unbound hidden value, use the HTML input tag with type 'hidden'.

<form:hidden path="house"/>

If we choose to submit the 'house' value as a hidden one, the HTML would look like:

<input name="house" type="hidden" value="Gryffindor"/>

The errors tag

This tag renders field errors in an HTML 'span' tag. It provides access to the errors created in your controller or those that were created by any validators associated with your controller.

Let’s assume we want to display all error messages for the firstName and lastName fields once we submit the form. We have a validator for instances of the User class called UserValidator.

public class UserValidator implements Validator {

public boolean supports(Class candidate) {

return User.class.isAssignableFrom(candidate);

}

public void validate(Object obj, Errors errors) {

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "required", "Field is required.");

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName", "required", "Field is required.");

}

}

The form.jsp would look like:

<form:form>

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName"/></td>

<%-- Show errors for firstName field --%>

<td><form:errors path="firstName"/></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName"/></td>

<%-- Show errors for lastName field --%>

<td><form:errors path="lastName"/></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form:form>

If we submit a form with empty values in the firstName and lastName fields, this is what the HTML would look like:

<form method="POST">

<table>

<tr>

<td>First Name:</td>

<td><input name="firstName" type="text" value=""/></td>

<%-- Associated errors to firstName field displayed --%>

<td><span name="firstName.errors">Field is required.</span></td>

</tr>

<tr>

<td>Last Name:</td>

<td><input name="lastName" type="text" value=""/></td>

<%-- Associated errors to lastName field displayed --%>

<td><span name="lastName.errors">Field is required.</span></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form>

What if we want to display the entire list of errors for a given page? The example below shows that the errors tag also supports some basic wildcarding functionality.

path="\*" - displays all errors

path="lastName" - displays all errors associated with the lastName field

if path is omitted - object errors only are displayed

The example below will display a list of errors at the top of the page, followed by field-specific errors next to the fields:

<form:form>

<form:errors path="\*" cssClass="errorBox"/>

<table>

<tr>

<td>First Name:</td>

<td><form:input path="firstName"/></td>

<td><form:errors path="firstName"/></td>

</tr>

<tr>

<td>Last Name:</td>

<td><form:input path="lastName"/></td>

<td><form:errors path="lastName"/></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form:form>

The HTML would look like:

<form method="POST">

<span name="\*.errors" class="errorBox">Field is required.<br/>Field is required.</span>

<table>

<tr>

<td>First Name:</td>

<td><input name="firstName" type="text" value=""/></td>

<td><span name="firstName.errors">Field is required.</span></td>

</tr>

<tr>

<td>Last Name:</td>

<td><input name="lastName" type="text" value=""/></td>

<td><span name="lastName.errors">Field is required.</span></td>

</tr>

<tr>

<td colspan="3">

<input type="submit" value="Save Changes"/>

</td>

</tr>

</table>

</form>

The spring-form.tld tag library descriptor (TLD) is included in the spring-webmvc.jar. For a comprehensive reference on individual tags, browse the API reference or see the tag library description.

HTTP method conversion

A key principle of REST is the use of the Uniform Interface. This means that all resources (URLs) can be manipulated using the same four HTTP methods: GET, PUT, POST, and DELETE. For each method, the HTTP specification defines the exact semantics. For instance, a GET should always be a safe operation, meaning that is has no side effects, and a PUT or DELETE should be idempotent, meaning that you can repeat these operations over and over again, but the end result should be the same. While HTTP defines these four methods, HTML only supports two: GET and POST. Fortunately, there are two possible workarounds: you can either use JavaScript to do your PUT or DELETE, or simply do a POST with the 'real' method as an additional parameter (modeled as a hidden input field in an HTML form). This latter trick is what Spring’s HiddenHttpMethodFilter does. This filter is a plain Servlet Filter and therefore it can be used in combination with any web framework (not just Spring MVC). Simply add this filter to your web.xml, and a POST with a hidden \_method parameter will be converted into the corresponding HTTP method request.

To support HTTP method conversion the Spring MVC form tag was updated to support setting the HTTP method. For example, the following snippet taken from the updated Petclinic sample

<form:form method="delete">

<p class="submit"><input type="submit" value="Delete Pet"/></p>

</form:form>

This will actually perform an HTTP POST, with the 'real' DELETE method hidden behind a request parameter, to be picked up by the HiddenHttpMethodFilter, as defined in web.xml:

<filter>

<filter-name>httpMethodFilter</filter-name>

<filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>httpMethodFilter</filter-name>

<servlet-name>petclinic</servlet-name>

</filter-mapping>

The corresponding @Controller method is shown below:

@RequestMapping(method = RequestMethod.DELETE)

public String deletePet(@PathVariable int ownerId, @PathVariable int petId) {

this.clinic.deletePet(petId);

return "redirect:/owners/" + ownerId;

}

HTML5 tags

Starting with Spring 3, the Spring form tag library allows entering dynamic attributes, which means you can enter any HTML5 specific attributes.

In Spring 3.1, the form input tag supports entering a type attribute other than 'text'. This is intended to allow rendering new HTML5 specific input types such as 'email', 'date', 'range', and others. Note that entering type='text' is not required since 'text' is the default type.

1.10.6. Tiles

It is possible to integrate Tiles - just as any other view technology - in web applications using Spring. The following describes in a broad way how to do this.

This section focuses on Spring’s support for Tiles v3 in the org.springframework.web.servlet.view.tiles3 package.

Dependencies

To be able to use Tiles, you have to add a dependency on Tiles version 3.0.1 or higher and its transitive dependencies to your project.

Configuration

To be able to use Tiles, you have to configure it using files containing definitions (for basic information on definitions and other Tiles concepts, please have a look at http://tiles.apache.org). In Spring this is done using the TilesConfigurer. Have a look at the following piece of example ApplicationContext configuration:

<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">

<property name="definitions">

<list>

<value>/WEB-INF/defs/general.xml</value>

<value>/WEB-INF/defs/widgets.xml</value>

<value>/WEB-INF/defs/administrator.xml</value>

<value>/WEB-INF/defs/customer.xml</value>

<value>/WEB-INF/defs/templates.xml</value>

</list>

</property>

</bean>

As you can see, there are five files containing definitions, which are all located in the 'WEB-INF/defs' directory. At initialization of the WebApplicationContext, the files will be loaded and the definitions factory will be initialized. After that has been done, the Tiles includes in the definition files can be used as views within your Spring web application. To be able to use the views you have to have a ViewResolver just as with any other view technology used with Spring. Below you can find two possibilities, the UrlBasedViewResolver and the ResourceBundleViewResolver.

You can specify locale specific Tiles definitions by adding an underscore and then the locale. For example:

<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">

<property name="definitions">

<list>

<value>/WEB-INF/defs/tiles.xml</value>

<value>/WEB-INF/defs/tiles\_fr\_FR.xml</value>

</list>

</property>

</bean>

With this configuration, tiles\_fr\_FR.xml will be used for requests with the fr\_FR locale, and tiles.xml will be used by default.

Since underscores are used to indicate locales, it is recommended to avoid using them otherwise in the file names for Tiles definitions.

UrlBasedViewResolver

The UrlBasedViewResolver instantiates the given viewClass for each view it has to resolve.

<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">

<property name="viewClass" value="org.springframework.web.servlet.view.tiles3.TilesView"/>

</bean>

ResourceBundleViewResolver

The ResourceBundleViewResolver has to be provided with a property file containing view names and view classes the resolver can use:

<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

<property name="basename" value="views"/>

</bean>

...

welcomeView.(class)=org.springframework.web.servlet.view.tiles3.TilesView

welcomeView.url=welcome (this is the name of a Tiles definition)

vetsView.(class)=org.springframework.web.servlet.view.tiles3.TilesView

vetsView.url=vetsView (again, this is the name of a Tiles definition)

findOwnersForm.(class)=org.springframework.web.servlet.view.JstlView

findOwnersForm.url=/WEB-INF/jsp/findOwners.jsp

...

As you can see, when using the ResourceBundleViewResolver, you can easily mix different view technologies.

Note that the TilesView class supports JSTL (the JSP Standard Tag Library) out of the box.

SimpleSpringPreparerFactory and SpringBeanPreparerFactory

As an advanced feature, Spring also supports two special Tiles PreparerFactory implementations. Check out the Tiles documentation for details on how to use ViewPreparer references in your Tiles definition files.

Specify SimpleSpringPreparerFactory to autowire ViewPreparer instances based on specified preparer classes, applying Spring’s container callbacks as well as applying configured Spring BeanPostProcessors. If Spring’s context-wide annotation-config has been activated, annotations in ViewPreparer classes will be automatically detected and applied. Note that this expects preparer classes in the Tiles definition files, just like the default PreparerFactory does.

Specify SpringBeanPreparerFactory to operate on specified preparer names instead of classes, obtaining the corresponding Spring bean from the DispatcherServlet’s application context. The full bean creation process will be in the control of the Spring application context in this case, allowing for the use of explicit dependency injection configuration, scoped beans etc. Note that you need to define one Spring bean definition per preparer name (as used in your Tiles definitions).

<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">

<property name="definitions">

<list>

<value>/WEB-INF/defs/general.xml</value>

<value>/WEB-INF/defs/widgets.xml</value>

<value>/WEB-INF/defs/administrator.xml</value>

<value>/WEB-INF/defs/customer.xml</value>

<value>/WEB-INF/defs/templates.xml</value>

</list>

</property>

<!-- resolving preparer names as Spring bean definition names -->

<property name="preparerFactoryClass"

value="org.springframework.web.servlet.view.tiles3.SpringBeanPreparerFactory"/>

</bean>

1.10.7. RSS, Atom

Both AbstractAtomFeedView and AbstractRssFeedView inherit from the base class AbstractFeedView and are used to provide Atom and RSS Feed views respectfully. They are based on java.net’s ROME project and are located in the package org.springframework.web.servlet.view.feed.

AbstractAtomFeedView requires you to implement the buildFeedEntries() method and optionally override the buildFeedMetadata() method (the default implementation is empty), as shown below.

public class SampleContentAtomView extends AbstractAtomFeedView {

@Override

protected void buildFeedMetadata(Map<String, Object> model,

Feed feed, HttpServletRequest request) {

// implementation omitted

}

@Override

protected List<Entry> buildFeedEntries(Map<String, Object> model,

HttpServletRequest request, HttpServletResponse response) throws Exception {

// implementation omitted

}

}

Similar requirements apply for implementing AbstractRssFeedView, as shown below.

public class SampleContentAtomView extends AbstractRssFeedView {

@Override

protected void buildFeedMetadata(Map<String, Object> model,

Channel feed, HttpServletRequest request) {

// implementation omitted

}

@Override

protected List<Item> buildFeedItems(Map<String, Object> model,

HttpServletRequest request, HttpServletResponse response) throws Exception {

// implementation omitted

}

}

The buildFeedItems() and buildFeedEntires() methods pass in the HTTP request in case you need to access the Locale. The HTTP response is passed in only for the setting of cookies or other HTTP headers. The feed will automatically be written to the response object after the method returns.

For an example of creating an Atom view please refer to Alef Arendsen’s Spring Team Blog entry.

1.10.8. PDF, Excel

Introduction

Returning an HTML page isn’t always the best way for the user to view the model output, and Spring makes it simple to generate a PDF document or an Excel spreadsheet dynamically from the model data. The document is the view and will be streamed from the server with the correct content type to (hopefully) enable the client PC to run their spreadsheet or PDF viewer application in response.

In order to use Excel views, you need to add the Apache POI library to your classpath, and for PDF generation preferably the OpenPDF library.

Use the latest versions of the underlying document generation libraries if possible. In particular, we strongly recommend OpenPDF (e.g. OpenPDF 1.0.5) instead of the outdated original iText 2.1.7 since it is actively maintained and fixes an important vulnerability for untrusted PDF content.

Configuration

Document based views are handled in an almost identical fashion to XSLT views, and the following sections build upon the previous one by demonstrating how the same controller used in the XSLT example is invoked to render the same model as both a PDF document and an Excel spreadsheet (which can also be viewed or manipulated in Open Office).

View definition

First, let’s amend the views.properties file (or xml equivalent) and add a simple view definition for both document types. The entire file now looks like this with the XSLT view shown from earlier:

home.(class)=xslt.HomePage

home.stylesheetLocation=/WEB-INF/xsl/home.xslt

home.root=words

xl.(class)=excel.HomePage

pdf.(class)=pdf.HomePage

If you want to start with a template spreadsheet or a fillable PDF form to add your model data to, specify the location as the 'url' property in the view definition

Controller

The controller code we’ll use remains exactly the same from the XSLT example earlier other than to change the name of the view to use. Of course, you could be clever and have this selected based on a URL parameter or some other logic - proof that Spring really is very good at decoupling the views from the controllers!

Excel views

Exactly as we did for the XSLT example, we’ll subclass suitable abstract classes in order to implement custom behavior in generating our output documents. For Excel, this involves writing a subclass of org.springframework.web.servlet.view.document.AbstractExcelView (for Excel files generated by POI) or org.springframework.web.servlet.view.document.AbstractJExcelView (for JExcelApi-generated Excel files) and implementing the buildExcelDocument() method.

Here’s the complete listing for our POI Excel view which displays the word list from the model map in consecutive rows of the first column of a new spreadsheet:

package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

protected void buildExcelDocument(Map model, HSSFWorkbook wb, HttpServletRequest req,

HttpServletResponse resp) throws Exception {

HSSFSheet sheet;

HSSFRow sheetRow;

HSSFCell cell;

// Go to the first sheet

// getSheetAt: only if wb is created from an existing document

// sheet = wb.getSheetAt(0);

sheet = wb.createSheet("Spring");

sheet.setDefaultColumnWidth((short) 12);

// write a text at A1

cell = getCell(sheet, 0, 0);

setText(cell, "Spring-Excel test");

List words = (List) model.get("wordList");

for (int i=0; i < words.size(); i++) {

cell = getCell(sheet, 2+i, 0);

setText(cell, (String) words.get(i));

}

}

}

And the following is a view generating the same Excel file, now using JExcelApi:

package excel;

// imports omitted for brevity

public class HomePage extends AbstractJExcelView {

protected void buildExcelDocument(Map model, WritableWorkbook wb,

HttpServletRequest request, HttpServletResponse response) throws Exception {

WritableSheet sheet = wb.createSheet("Spring", 0);

sheet.addCell(new Label(0, 0, "Spring-Excel test"));

List words = (List) model.get("wordList");

for (int i = 0; i < words.size(); i++) {

sheet.addCell(new Label(2+i, 0, (String) words.get(i)));

}

}

}

Note the differences between the APIs. We’ve found that the JExcelApi is somewhat more intuitive, and furthermore, JExcelApi has slightly better image-handling capabilities. There have been memory problems with large Excel files when using JExcelApi however.

If you now amend the controller such that it returns xl as the name of the view ( return new ModelAndView("xl", map);) and run your application again, you should find that the Excel spreadsheet is created and downloaded automatically when you request the same page as before.

PDF views

The PDF version of the word list is even simpler. This time, the class extends org.springframework.web.servlet.view.document.AbstractPdfView and implements the buildPdfDocument() method as follows:

package pdf;

// imports omitted for brevity

public class PDFPage extends AbstractPdfView {

protected void buildPdfDocument(Map model, Document doc, PdfWriter writer,

HttpServletRequest req, HttpServletResponse resp) throws Exception {

List words = (List) model.get("wordList");

for (int i=0; i<words.size(); i++) {

doc.add( new Paragraph((String) words.get(i)));

}

}

}

Once again, amend the controller to return the pdf view with return new ModelAndView("pdf", map);, and reload the URL in your application. This time a PDF document should appear listing each of the words in the model map.

1.10.9. Jackson

Same in Spring WebFlux

JSON

Same in Spring WebFlux

The MappingJackson2JsonView uses the Jackson library’s ObjectMapper to render the response content as JSON. By default, the entire contents of the model map (with the exception of framework-specific classes) will be encoded as JSON. For cases where the contents of the map need to be filtered, users may specify a specific set of model attributes to encode via the RenderedAttributes property. The extractValueFromSingleKeyModel property may also be used to have the value in single-key models extracted and serialized directly rather than as a map of model attributes.

JSON mapping can be customized as needed through the use of Jackson’s provided annotations. When further control is needed, a custom ObjectMapper can be injected through the ObjectMapper property for cases where custom JSON serializers/deserializers need to be provided for specific types.

As of Spring Framework 5.0.7, JSONP support is deprecated and requires to customize the JSONP query parameter name(s) through the jsonpParameterNames property. This support will be removed as of Spring Framework 5.1, CORS should be used instead.

XML

Same in Spring WebFlux

The MappingJackson2XmlView uses the Jackson XML extension's XmlMapper to render the response content as XML. If the model contains multiples entries, the object to be serialized should be set explicitly using the modelKey bean property. If the model contains a single entry, it will be serialized automatically.

XML mapping can be customized as needed through the use of JAXB or Jackson’s provided annotations. When further control is needed, a custom XmlMapper can be injected through the ObjectMapper property for cases where custom XML serializers/deserializers need to be provided for specific types.

1.10.10. XML

The MarshallingView uses an XML Marshaller defined in the org.springframework.oxm package to render the response content as XML. The object to be marshalled can be set explicitly using MarhsallingView's modelKey bean property. Alternatively, the view will iterate over all model properties and marshal the first type that is supported by the Marshaller. For more information on the functionality in the org.springframework.oxm package refer to the chapter Marshalling XML using O/X Mappers.

1.10.11. XSLT

XSLT is a transformation language for XML and is popular as a view technology within web applications. XSLT can be a good choice as a view technology if your application naturally deals with XML, or if your model can easily be converted to XML. The following section shows how to produce an XML document as model data and have it transformed with XSLT in a Spring Web MVC application.

This example is a trivial Spring application that creates a list of words in the Controller and adds them to the model map. The map is returned along with the view name of our XSLT view. See Annotated Controllers for details of Spring Web MVC’s Controller interface. The XSLT Controller will turn the list of words into a simple XML document ready for transformation.

Beans

Configuration is standard for a simple Spring application. The MVC configuration has to define a XsltViewResolver bean and regular MVC annotation configuration.

@EnableWebMvc

@ComponentScan

@Configuration

public class WebConfig implements WebMvcConfigurer {

@Bean

public XsltViewResolver xsltViewResolver() {

XsltViewResolver viewResolver = new XsltViewResolver();

viewResolver.setPrefix("/WEB-INF/xsl/");

viewResolver.setSuffix(".xslt");

return viewResolver;

}

}

And we need a Controller that encapsulates our word generation logic.

Controller

The controller logic is encapsulated in a @Controller class, with the handler method being defined like so…​

@Controller

public class XsltController {

@RequestMapping("/")

public String home(Model model) throws Exception {

Document document = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();

Element root = document.createElement("wordList");

List<String> words = Arrays.asList("Hello", "Spring", "Framework");

for (String word : words) {

Element wordNode = document.createElement("word");

Text textNode = document.createTextNode(word);

wordNode.appendChild(textNode);

root.appendChild(wordNode);

}

model.addAttribute("wordList", root);

return "home";

}

}

So far we’ve only created a DOM document and added it to the Model map. Note that you can also load an XML file as a Resource and use it instead of a custom DOM document.

Of course, there are software packages available that will automatically 'domify' an object graph, but within Spring, you have complete flexibility to create the DOM from your model in any way you choose. This prevents the transformation of XML playing too great a part in the structure of your model data which is a danger when using tools to manage the domification process.

Next, XsltViewResolver will resolve the "home" XSLT template file and merge the DOM document into it to generate our view.

Transformation

Finally, the XsltViewResolver will resolve the "home" XSLT template file and merge the DOM document into it to generate our view. As shown in the XsltViewResolver configuration, XSLT templates live in the war file in the 'WEB-INF/xsl' directory and end with a "xslt" file extension.

<?xml version="1.0" encoding="utf-8"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="html" omit-xml-declaration="yes"/>

<xsl:template match="/">

<html>

<head><title>Hello!</title></head>

<body>

<h1>My First Words</h1>

<ul>

<xsl:apply-templates/>

</ul>

</body>

</html>

</xsl:template>

<xsl:template match="word">

<li><xsl:value-of select="."/></li>

</xsl:template>

</xsl:stylesheet>

This is rendered as:

<html>

<head>

<META http-equiv="Content-Type" content="text/html; charset=UTF-8">

<title>Hello!</title>

</head>

<body>

<h1>My First Words</h1>

<ul>

<li>Hello</li>

<li>Spring</li>

<li>Framework</li>

</ul>

</body>

</html>

1.11. MVC Config

Same in Spring WebFlux

The MVC Java config and the MVC XML namespace provide default configuration suitable for most applications along with a configuration API to customize it.

For more advanced customizations, not available in the configuration API, see Advanced Java Config and Advanced XML Config.

You do not need to understand the underlying beans created by the MVC Java config and the MVC namespace but if you want to learn more, see Special Bean Types and Web MVC Config.

1.11.1. Enable MVC Config

Same in Spring WebFlux

In Java config use the @EnableWebMvc annotation:

@Configuration

@EnableWebMvc

public class WebConfig {

}

In XML use the <mvc:annotation-driven> element:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:mvc="http://www.springframework.org/schema/mvc"

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

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd">

<mvc:annotation-driven/>

</beans>

The above registers a number of Spring MVC infrastructure beans also adapting to dependencies available on the classpath: e.g. payload converters for JSON, XML, etc.

1.11.2. MVC Config API

Same in Spring WebFlux

In Java config implement WebMvcConfigurer interface:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

// Implement configuration methods...

}

In XML check attributes and sub-elements of <mvc:annotation-driven/>. You can view the Spring MVC XML schema or use the code completion feature of your IDE to discover what attributes and sub-elements are available.

1.11.3. Type conversion

Same in Spring WebFlux

By default formatters for Number and Date types are installed, including support for the @NumberFormat and @DateTimeFormat annotations. Full support for the Joda-Time formatting library is also installed if Joda-Time is present on the classpath.

In Java config, register custom formatters and converters:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void addFormatters(FormatterRegistry registry) {

// ...

}

}

In XML, the same:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:mvc="http://www.springframework.org/schema/mvc"

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

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd">

<mvc:annotation-driven conversion-service="conversionService"/>

<bean id="conversionService"

class="org.springframework.format.support.FormattingConversionServiceFactoryBean">

<property name="converters">

<set>

<bean class="org.example.MyConverter"/>

</set>

</property>

<property name="formatters">

<set>

<bean class="org.example.MyFormatter"/>

<bean class="org.example.MyAnnotationFormatterFactory"/>

</set>

</property>

<property name="formatterRegistrars">

<set>

<bean class="org.example.MyFormatterRegistrar"/>

</set>

</property>

</bean>

</beans>

See FormatterRegistrar SPI and the FormattingConversionServiceFactoryBean for more information on when to use FormatterRegistrars.

1.11.4. Validation

Same in Spring WebFlux

By default if Bean Validation is present on the classpath — e.g. Hibernate Validator, the LocalValidatorFactoryBean is registered as a global Validator for use with @Valid and Validated on controller method arguments.

In Java config, you can customize the global Validator instance:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public Validator getValidator(); {

// ...

}

}

In XML, the same:

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:mvc="http://www.springframework.org/schema/mvc"

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

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd">

<mvc:annotation-driven validator="globalValidator"/>

</beans>

Note that you can also register Validator's locally:

@Controller

public class MyController {

@InitBinder

protected void initBinder(WebDataBinder binder) {

binder.addValidators(new FooValidator());

}

}

If you need to have a LocalValidatorFactoryBean injected somewhere, create a bean and mark it with @Primary in order to avoid conflict with the one declared in the MVC config.

1.11.5. Interceptors

In Java config, register interceptors to apply to incoming requests:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(new LocaleChangeInterceptor());

registry.addInterceptor(new ThemeChangeInterceptor()).addPathPatterns("/\*\*").excludePathPatterns("/admin/\*\*");

registry.addInterceptor(new SecurityInterceptor()).addPathPatterns("/secure/\*");

}

}

In XML, the same:

<mvc:interceptors>

<bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"/>

<mvc:interceptor>

<mvc:mapping path="/\*\*"/>

<mvc:exclude-mapping path="/admin/\*\*"/>

<bean class="org.springframework.web.servlet.theme.ThemeChangeInterceptor"/>

</mvc:interceptor>

<mvc:interceptor>

<mvc:mapping path="/secure/\*"/>

<bean class="org.example.SecurityInterceptor"/>

</mvc:interceptor>

</mvc:interceptors>

1.11.6. Content Types

Same in Spring WebFlux

You can configure how Spring MVC determines the requested media types from the request — e.g. Accept header, URL path extension, query parameter, etc.

By default the URL path extension is checked first — with json, xml, rss, and atom registered as known extensions depending on classpath dependencies, and the "Accept" header is checked second.

Consider changing those defaults to Accept header only and if you must use URL-based content type resolution consider the query parameter strategy over the path extensions. See Suffix match and Suffix match and RFD for more details.

In Java config, customize requested content type resolution:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {

configurer.mediaType("json", MediaType.APPLICATION\_JSON);

configurer.mediaType("xml", MediaType.APPLICATION\_XML);

}

}

In XML, the same:

<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager"/>

<bean id="contentNegotiationManager" class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">

<property name="mediaTypes">

<value>

json=application/json

xml=application/xml

</value>

</property>

</bean>

1.11.7. Message Converters

Same in Spring WebFlux

Customization of HttpMessageConverter can be achieved in Java config by overriding configureMessageConverters() if you want to replace the default converters created by Spring MVC, or by overriding extendMessageConverters() if you just want to customize them or add additional converters to the default ones.

Below is an example that adds Jackson JSON and XML converters with a customized ObjectMapper instead of default ones:

@Configuration

@EnableWebMvc

public class WebConfiguration implements WebMvcConfigurer {

@Override

public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {

Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder()

.indentOutput(true)

.dateFormat(new SimpleDateFormat("yyyy-MM-dd"))

.modulesToInstall(new ParameterNamesModule());

converters.add(new MappingJackson2HttpMessageConverter(builder.build()));

converters.add(new MappingJackson2XmlHttpMessageConverter(builder.createXmlMapper(true).build()));

}

}

In this example, Jackson2ObjectMapperBuilder is used to create a common configuration for both MappingJackson2HttpMessageConverter and MappingJackson2XmlHttpMessageConverter with indentation enabled, a customized date format and the registration of jackson-module-parameter-names that adds support for accessing parameter names (feature added in Java 8).

This builder customizes Jackson’s default properties with the following ones:

DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES is disabled.

MapperFeature.DEFAULT\_VIEW\_INCLUSION is disabled.

It also automatically registers the following well-known modules if they are detected on the classpath:

jackson-datatype-jdk7: support for Java 7 types like java.nio.file.Path.

jackson-datatype-joda: support for Joda-Time types.

jackson-datatype-jsr310: support for Java 8 Date & Time API types.

jackson-datatype-jdk8: support for other Java 8 types like Optional.

Enabling indentation with Jackson XML support requires woodstox-core-asl dependency in addition to jackson-dataformat-xml one.

Other interesting Jackson modules are available:

jackson-datatype-money: support for javax.money types (unofficial module)

jackson-datatype-hibernate: support for Hibernate specific types and properties (including lazy-loading aspects)

It is also possible to do the same in XML:

<mvc:annotation-driven>

<mvc:message-converters>

<bean class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter">

<property name="objectMapper" ref="objectMapper"/>

</bean>

<bean class="org.springframework.http.converter.xml.MappingJackson2XmlHttpMessageConverter">

<property name="objectMapper" ref="xmlMapper"/>

</bean>

</mvc:message-converters>

</mvc:annotation-driven>

<bean id="objectMapper" class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean"

p:indentOutput="true"

p:simpleDateFormat="yyyy-MM-dd"

p:modulesToInstall="com.fasterxml.jackson.module.paramnames.ParameterNamesModule"/>

<bean id="xmlMapper" parent="objectMapper" p:createXmlMapper="true"/>

1.11.8. View Controllers

This is a shortcut for defining a ParameterizableViewController that immediately forwards to a view when invoked. Use it in static cases when there is no Java controller logic to execute before the view generates the response.

An example of forwarding a request for "/" to a view called "home" in Java:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void addViewControllers(ViewControllerRegistry registry) {

registry.addViewController("/").setViewName("home");

}

}

And the same in XML use the <mvc:view-controller> element:

<mvc:view-controller path="/" view-name="home"/>

1.11.9. View Resolvers

Same in Spring WebFlux

The MVC config simplifies the registration of view resolvers.

The following is a Java config example that configures content negotiation view resolution using JSP and Jackson as a default View for JSON rendering:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.enableContentNegotiation(new MappingJackson2JsonView());

registry.jsp();

}

}

And the same in XML:

<mvc:view-resolvers>

<mvc:content-negotiation>

<mvc:default-views>

<bean class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>

</mvc:default-views>

</mvc:content-negotiation>

<mvc:jsp/>

</mvc:view-resolvers>

Note however that FreeMarker, Tiles, Groovy Markup and script templates also require configuration of the underlying view technology.

The MVC namespace provides dedicated elements. For example with FreeMarker:

<mvc:view-resolvers>

<mvc:content-negotiation>

<mvc:default-views>

<bean class="org.springframework.web.servlet.view.json.MappingJackson2JsonView"/>

</mvc:default-views>

</mvc:content-negotiation>

<mvc:freemarker cache="false"/>

</mvc:view-resolvers>

<mvc:freemarker-configurer>

<mvc:template-loader-path location="/freemarker"/>

</mvc:freemarker-configurer>

In Java config simply add the respective "Configurer" bean:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.enableContentNegotiation(new MappingJackson2JsonView());

registry.freeMarker().cache(false);

}

@Bean

public FreeMarkerConfigurer freeMarkerConfigurer() {

FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();

configurer.setTemplateLoaderPath("/freemarker");

return configurer;

}

}

1.11.10. Static Resources

Same in Spring WebFlux

This option provides a convenient way to serve static resources from a list of Resource-based locations.

In the example below, given a request that starts with "/resources", the relative path is used to find and serve static resources relative to "/public" under the web application root or on the classpath under "/static". The resources are served with a 1-year future expiration to ensure maximum use of the browser cache and a reduction in HTTP requests made by the browser. The Last-Modified header is also evaluated and if present a 304 status code is returned.

In Java config:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*")

.addResourceLocations("/public", "classpath:/static/")

.setCachePeriod(31556926);

}

}

In XML:

<mvc:resources mapping="/resources/\*\*"

location="/public, classpath:/static/"

cache-period="31556926" />

See also HTTP caching support for static resources.

The resource handler also supports a chain of ResourceResolvers and ResourceTransformers. which can be used to create a toolchain for working with optimized resources.

The VersionResourceResolver can be used for versioned resource URLs based on an MD5 hash computed from the content, a fixed application version, or other. A ContentVersionStrategy (MD5 hash) is a good choice with some notable exceptions such as JavaScript resources used with a module loader.

For example in Java config;

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*")

.addResourceLocations("/public/")

.resourceChain(true)

.addResolver(new VersionResourceResolver().addContentVersionStrategy("/\*\*"));

}

}

In XML, the same:

<mvc:resources mapping="/resources/\*\*" location="/public/">

<mvc:resource-chain>

<mvc:resource-cache/>

<mvc:resolvers>

<mvc:version-resolver>

<mvc:content-version-strategy patterns="/\*\*"/>

</mvc:version-resolver>

</mvc:resolvers>

</mvc:resource-chain>

</mvc:resources>

You can use ResourceUrlProvider to rewrite URLs and apply the full chain of resolvers and transformers — e.g. to insert versions. The MVC config provides a ResourceUrlProvider bean so it can be injected into others. You can also make the rewrite transparent with the ResourceUrlEncodingFilter for Thymeleaf, JSPs, FreeMarker, and others with URL tags that rely on HttpServletResponse#encodeURL.

WebJars is also supported via WebJarsResourceResolver and automatically registered when "org.webjars:webjars-locator" is present on the classpath. The resolver can re-write URLs to include the version of the jar and can also match to incoming URLs without versions — e.g. "/jquery/jquery.min.js" to "/jquery/1.2.0/jquery.min.js".

1.11.11. Default Servlet

This allows for mapping the DispatcherServlet to "/" (thus overriding the mapping of the container’s default Servlet), while still allowing static resource requests to be handled by the container’s default Servlet. It configures a DefaultServletHttpRequestHandler with a URL mapping of "/\*\*" and the lowest priority relative to other URL mappings.

This handler will forward all requests to the default Servlet. Therefore it is important that it remains last in the order of all other URL HandlerMappings. That will be the case if you use <mvc:annotation-driven> or alternatively if you are setting up your own customized HandlerMapping instance be sure to set its order property to a value lower than that of the DefaultServletHttpRequestHandler, which is Integer.MAX\_VALUE.

To enable the feature using the default setup use:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

configurer.enable();

}

}

Or in XML:

<mvc:default-servlet-handler/>

The caveat to overriding the "/" Servlet mapping is that the RequestDispatcher for the default Servlet must be retrieved by name rather than by path. The DefaultServletHttpRequestHandler will attempt to auto-detect the default Servlet for the container at startup time, using a list of known names for most of the major Servlet containers (including Tomcat, Jetty, GlassFish, JBoss, Resin, WebLogic, and WebSphere). If the default Servlet has been custom configured with a different name, or if a different Servlet container is being used where the default Servlet name is unknown, then the default Servlet’s name must be explicitly provided as in the following example:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {

configurer.enable("myCustomDefaultServlet");

}

}

Or in XML:

<mvc:default-servlet-handler default-servlet-name="myCustomDefaultServlet"/>

1.11.12. Path Matching

Same in Spring WebFlux

This allows customizing options related to URL matching and treatment of the URL. For details on the individual options check out the PathMatchConfigurer API.

Example in Java config:

@Configuration

@EnableWebMvc

public class WebConfig implements WebMvcConfigurer {

@Override

public void configurePathMatch(PathMatchConfigurer configurer) {

configurer

.setUseSuffixPatternMatch(true)

.setUseTrailingSlashMatch(false)

.setUseRegisteredSuffixPatternMatch(true)

.setPathMatcher(antPathMatcher())

.setUrlPathHelper(urlPathHelper());

}

@Bean

public UrlPathHelper urlPathHelper() {

//...

}

@Bean

public PathMatcher antPathMatcher() {

//...

}

}

In XML, the same:

<mvc:annotation-driven>

<mvc:path-matching

suffix-pattern="true"

trailing-slash="false"

registered-suffixes-only="true"

path-helper="pathHelper"

path-matcher="pathMatcher"/>

</mvc:annotation-driven>

<bean id="pathHelper" class="org.example.app.MyPathHelper"/>

<bean id="pathMatcher" class="org.example.app.MyPathMatcher"/>

1.11.13. Advanced Java Config

Same in Spring WebFlux

@EnableWebMvc imports DelegatingWebMvcConfiguration that (1) provides default Spring configuration for Spring MVC applications and (2) detects and delegates to WebMvcConfigurer's to customize that configuration.

For advanced mode, remove @EnableWebMvc and extend directly from DelegatingWebMvcConfiguration instead of implementing WebMvcConfigurer:

@Configuration

public class WebConfig extends DelegatingWebMvcConfiguration {

// ...

}

You can keep existing methods in WebConfig but you can now also override bean declarations from the base class and you can still have any number of other WebMvcConfigurer's on the classpath.

1.11.14. Advanced XML Config

The MVC namespace does not have an advanced mode. If you need to customize a property on a bean that you can’t change otherwise, you can use the BeanPostProcessor lifecycle hook of the Spring ApplicationContext:

@Component

public class MyPostProcessor implements BeanPostProcessor {

public Object postProcessBeforeInitialization(Object bean, String name) throws BeansException {

// ...

}

}

Note that MyPostProcessor needs to be declared as a bean either explicitly in XML or detected through a <component-scan/> declaration.

1.12. HTTP/2

Same in Spring WebFlux

Servlet 4 containers are required to support HTTP/2 and Spring Framework 5 is compatible with Servlet API 4. From a programming model perspective there is nothing specific that applications need to do. However there are considerations related to server configuration. For more details please check out the HTTP/2 wiki page.

The Servlet API does expose one construct related to HTTP/2. The javax.servlet.http.PushBuilder can used to proactively push resources to clients and it is supported as a method argument to @RequestMapping methods.

2. REST Clients

This section describes options for client-side access to REST endpoints.

2.1. RestTemplate

RestTemplate is the original Spring REST client that follows a similar approach to other template classes in the Spring Framework (e.g. JdbcTemplate, JmsTemplate, etc.) by providing a list of parameterizable methods to perform HTTP requests.

RestTemplate has a synchronous API and relies on blocking I/O. This is okay for client scenarios with low concurrency. In a server environment or when orchestrating a sequence of remote calls, prefer using the WebClient which provides a more efficient execution model including seamless support for streaming.

See RestTemplate for more details on using the RestTemplate.

2.2. WebClient

WebClient is a reactive client that provides an alternative to the RestTemplate. It exposes a functional, fluent API and relies on non-blocking I/O which allows it to support high concurrency more efficiently (i.e. using a small number of threads) than the RestTemplate. WebClient is a natural fit for streaming scenarios.

See WebClient for more details on using the WebClient.

3. Testing

Same in Spring WebFlux

This section summarizes the options available in spring-test for Spring MVC applications.

Servlet API Mocks

Mock implementations of Servlet API contracts for unit testing controllers, filters, and other web components. See Servlet API mock objects for more details.

TestContext Framework

Support for loading Spring configuration in JUnit and TestNG tests including efficient caching of the loaded configuration across test methods and support for loading a WebApplicationContext with a MockServletContext. See TestContext Framework for more details.

Spring MVC Test

A framework, also known as MockMvc, for testing annotated controllers through the DispatcherServlet, i.e. supporting annotations and complete with Spring MVC infrastructure, but without an HTTP server. See Spring MVC Test for more details.

Client-side REST

spring-test provides a MockRestServiceServer that can be used as a mock server for testing client-side code that internally uses the RestTemplate. See Client REST Tests for more details.

WebTestClient

WebTestClient was built for testing WebFlux applications but it can also be used for end-to-end integration testing, to any server, over an HTTP connection. It is a non-blocking, reactive client and well suited for testing asynchronous and streaming scenarios.

4. WebSockets

Same in Spring WebFlux

This part of the reference documentation covers support for Servlet stack, WebSocket messaging that includes raw WebSocket interactions, WebSocket emulation via SockJS, and pub-sub messaging via STOMP as a sub-protocol over WebSocket.

4.1. Introduction

The WebSocket protocol RFC 6455 provides a standardized way to establish a full-duplex, two-way communication channel between client and server over a single TCP connection. It is a different TCP protocol from HTTP but is designed to work over HTTP, using ports 80 and 443 and allowing re-use of existing firewall rules.

A WebSocket interaction begins with an HTTP request that uses the HTTP "Upgrade" header to upgrade, or in this case to switch, to the WebSocket protocol:

GET /spring-websocket-portfolio/portfolio HTTP/1.1

Host: localhost:8080

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: Uc9l9TMkWGbHFD2qnFHltg==

Sec-WebSocket-Protocol: v10.stomp, v11.stomp

Sec-WebSocket-Version: 13

Origin: http://localhost:8080

Instead of the usual 200 status code, a server with WebSocket support returns:

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: 1qVdfYHU9hPOl4JYYNXF623Gzn0=

Sec-WebSocket-Protocol: v10.stomp

After a successful handshake the TCP socket underlying the HTTP upgrade request remains open for both client and server to continue to send and receive messages.

A complete introduction of how WebSockets work is beyond the scope of this document. Please read RFC 6455, the WebSocket chapter of HTML5, or one of many introductions and tutorials on the Web.

Note that if a WebSocket server is running behind a web server (e.g. nginx) you will likely need to configure it to pass WebSocket upgrade requests on to the WebSocket server. Likewise if the application runs in a cloud environment, check the instructions of the cloud provider related to WebSocket support.

4.1.1. HTTP vs WebSocket

Even though WebSocket is designed to be HTTP compatible and starts with an HTTP request, it is important to understand that the two protocols lead to very different architectures and application programming models.

In HTTP and REST, an application is modeled as many URLs. To interact with the application clients access those URLs, request-response style. Servers route requests to the appropriate handler based on the HTTP URL, method, and headers.

By contrast in WebSockets there is usually just one URL for the initial connect and subsequently all application messages flow on that same TCP connection. This points to an entirely different asynchronous, event-driven, messaging architecture.

WebSocket is also a low-level transport protocol which unlike HTTP does not prescribe any semantics to the content of messages. That means there is no way to route or process a message unless client and server agree on message semantics.

WebSocket clients and servers can negotiate the use of a higher-level, messaging protocol (e.g. STOMP), via the "Sec-WebSocket-Protocol" header on the HTTP handshake request, or in the absence of that they need to come up with their own conventions.

4.1.2. When to use it?

WebSockets can make a web page dynamic and interactive. However in many cases a combination of Ajax and HTTP streaming and/or long polling could provide a simple and effective solution.

For example news, mail, and social feeds need to update dynamically but it may be perfectly okay to do so every few minutes. Collaboration, games, and financial apps on the other hand need to be much closer to real time.

Latency alone is not a deciding factor. If the volume of messages is relatively low (e.g. monitoring network failures) HTTP streaming or polling may provide an effective solution. It is the combination of low latency, high frequency and high volume that make the best case for the use WebSocket.

Keep in mind also that over the Internet, restrictive proxies outside your control, may preclude WebSocket interactions either because they are not configured to pass on the Upgrade header or because they close long lived connections that appear idle? This means that the use of WebSocket for internal applications within the firewall is a more straight-forward decision than it is for public facing applications.

4.2. WebSocket API

Same in Spring WebFlux

The Spring Framework provides a WebSocket API that can be used to write client and server side applications that handle WebSocket messages.

4.2.1. WebSocketHandler

Same in Spring WebFlux

Creating a WebSocket server is as simple as implementing WebSocketHandler or more likely extending either TextWebSocketHandler or BinaryWebSocketHandler:

import org.springframework.web.socket.WebSocketHandler;

import org.springframework.web.socket.WebSocketSession;

import org.springframework.web.socket.TextMessage;

public class MyHandler extends TextWebSocketHandler {

@Override

public void handleTextMessage(WebSocketSession session, TextMessage message) {

// ...

}

}

There is dedicated WebSocket Java-config and XML namespace support for mapping the above WebSocket handler to a specific URL:

import org.springframework.web.socket.config.annotation.EnableWebSocket;

import org.springframework.web.socket.config.annotation.WebSocketConfigurer;

import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@Configuration

@EnableWebSocket

public class WebSocketConfig implements WebSocketConfigurer {

@Override

public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {

registry.addHandler(myHandler(), "/myHandler");

}

@Bean

public WebSocketHandler myHandler() {

return new MyHandler();

}

}

XML configuration equivalent:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:handlers>

<websocket:mapping path="/myHandler" handler="myHandler"/>

</websocket:handlers>

<bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>

The above is for use in Spring MVC applications and should be included in the configuration of a DispatcherServlet. However, Spring’s WebSocket support does not depend on Spring MVC. It is relatively simple to integrate a WebSocketHandler into other HTTP serving environments with the help of WebSocketHttpRequestHandler.

4.2.2. WebSocket Handshake

Same in Spring WebFlux

The easiest way to customize the initial HTTP WebSocket handshake request is through a HandshakeInterceptor, which exposes "before" and "after" the handshake methods. Such an interceptor can be used to preclude the handshake or to make any attributes available to the WebSocketSession. For example, there is a built-in interceptor for passing HTTP session attributes to the WebSocket session:

@Configuration

@EnableWebSocket

public class WebSocketConfig implements WebSocketConfigurer {

@Override

public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {

registry.addHandler(new MyHandler(), "/myHandler")

.addInterceptors(new HttpSessionHandshakeInterceptor());

}

}

And the XML configuration equivalent:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:handlers>

<websocket:mapping path="/myHandler" handler="myHandler"/>

<websocket:handshake-interceptors>

<bean class="org.springframework.web.socket.server.support.HttpSessionHandshakeInterceptor"/>

</websocket:handshake-interceptors>

</websocket:handlers>

<bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>

A more advanced option is to extend the DefaultHandshakeHandler that performs the steps of the WebSocket handshake, including validating the client origin, negotiating a sub-protocol, and others. An application may also need to use this option if it needs to configure a custom RequestUpgradeStrategy in order to adapt to a WebSocket server engine and version that is not yet supported (also see Deployment for more on this subject). Both the Java-config and XML namespace make it possible to configure a custom HandshakeHandler.

Spring provides a WebSocketHandlerDecorator base class that can be used to decorate a WebSocketHandler with additional behavior. Logging and exception handling implementations are provided and added by default when using the WebSocket Java-config or XML namespace. The ExceptionWebSocketHandlerDecorator catches all uncaught exceptions arising from any WebSocketHandler method and closes the WebSocket session with status 1011 that indicates a server error.

4.2.3. Deployment

The Spring WebSocket API is easy to integrate into a Spring MVC application where the DispatcherServlet serves both HTTP WebSocket handshake as well as other HTTP requests. It is also easy to integrate into other HTTP processing scenarios by invoking WebSocketHttpRequestHandler. This is convenient and easy to understand. However, special considerations apply with regards to JSR-356 runtimes.

The Java WebSocket API (JSR-356) provides two deployment mechanisms. The first involves a Servlet container classpath scan (Servlet 3 feature) at startup; and the other is a registration API to use at Servlet container initialization. Neither of these mechanism makes it possible to use a single "front controller" for all HTTP processing — including WebSocket handshake and all other HTTP requests — such as Spring MVC’s DispatcherServlet.

This is a significant limitation of JSR-356 that Spring’s WebSocket support addresses server-specific RequestUpgradeStrategy's even when running in a JSR-356 runtime. Such strategies currently exist for Tomcat, Jetty, GlassFish, WebLogic, WebSphere, and Undertow (and WildFly).

A request to overcome the above limitation in the Java WebSocket API has been created and can be followed at WEBSOCKET\_SPEC-211. Tomcat, Undertow and WebSphere provide their own API alternatives that makes it possible to this, and it’s also possible with Jetty. We are hopeful that more servers will follow do the same.

A secondary consideration is that Servlet containers with JSR-356 support are expected to perform a ServletContainerInitializer (SCI) scan that can slow down application startup, in some cases dramatically. If a significant impact is observed after an upgrade to a Servlet container version with JSR-356 support, it should be possible to selectively enable or disable web fragments (and SCI scanning) through the use of the <absolute-ordering /> element in web.xml:

<web-app xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"

version="3.0">

<absolute-ordering/>

</web-app>

You can then selectively enable web fragments by name, such as Spring’s own SpringServletContainerInitializer that provides support for the Servlet 3 Java initialization API, if required:

<web-app xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"

version="3.0">

<absolute-ordering>

<name>spring\_web</name>

</absolute-ordering>

</web-app>

4.2.4. Server config

Same in Spring WebFlux

Each underlying WebSocket engine exposes configuration properties that control runtime characteristics such as the size of message buffer sizes, idle timeout, and others.

For Tomcat, WildFly, and GlassFish add a ServletServerContainerFactoryBean to your WebSocket Java config:

@Configuration

@EnableWebSocket

public class WebSocketConfig implements WebSocketConfigurer {

@Bean

public ServletServerContainerFactoryBean createWebSocketContainer() {

ServletServerContainerFactoryBean container = new ServletServerContainerFactoryBean();

container.setMaxTextMessageBufferSize(8192);

container.setMaxBinaryMessageBufferSize(8192);

return container;

}

}

or WebSocket XML namespace:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<bean class="org.springframework...ServletServerContainerFactoryBean">

<property name="maxTextMessageBufferSize" value="8192"/>

<property name="maxBinaryMessageBufferSize" value="8192"/>

</bean>

</beans>

For client side WebSocket configuration, you should use WebSocketContainerFactoryBean (XML) or ContainerProvider.getWebSocketContainer() (Java config).

For Jetty, you’ll need to supply a pre-configured Jetty WebSocketServerFactory and plug that into Spring’s DefaultHandshakeHandler through your WebSocket Java config:

@Configuration

@EnableWebSocket

public class WebSocketConfig implements WebSocketConfigurer {

@Override

public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {

registry.addHandler(echoWebSocketHandler(),

"/echo").setHandshakeHandler(handshakeHandler());

}

@Bean

public DefaultHandshakeHandler handshakeHandler() {

WebSocketPolicy policy = new WebSocketPolicy(WebSocketBehavior.SERVER);

policy.setInputBufferSize(8192);

policy.setIdleTimeout(600000);

return new DefaultHandshakeHandler(

new JettyRequestUpgradeStrategy(new WebSocketServerFactory(policy)));

}

}

or WebSocket XML namespace:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:handlers>

<websocket:mapping path="/echo" handler="echoHandler"/>

<websocket:handshake-handler ref="handshakeHandler"/>

</websocket:handlers>

<bean id="handshakeHandler" class="org.springframework...DefaultHandshakeHandler">

<constructor-arg ref="upgradeStrategy"/>

</bean>

<bean id="upgradeStrategy" class="org.springframework...JettyRequestUpgradeStrategy">

<constructor-arg ref="serverFactory"/>

</bean>

<bean id="serverFactory" class="org.eclipse.jetty...WebSocketServerFactory">

<constructor-arg>

<bean class="org.eclipse.jetty...WebSocketPolicy">

<constructor-arg value="SERVER"/>

<property name="inputBufferSize" value="8092"/>

<property name="idleTimeout" value="600000"/>

</bean>

</constructor-arg>

</bean>

</beans>

4.2.5. Allowed origins

Same in Spring WebFlux

As of Spring Framework 4.1.5, the default behavior for WebSocket and SockJS is to accept only same origin requests. It is also possible to allow all or a specified list of origins. This check is mostly designed for browser clients. There is nothing preventing other types of clients from modifying the Origin header value (see RFC 6454: The Web Origin Concept for more details).

The 3 possible behaviors are:

Allow only same origin requests (default): in this mode, when SockJS is enabled, the Iframe HTTP response header X-Frame-Options is set to SAMEORIGIN, and JSONP transport is disabled since it does not allow to check the origin of a request. As a consequence, IE6 and IE7 are not supported when this mode is enabled.

Allow a specified list of origins: each provided allowed origin must start with http:// or https://. In this mode, when SockJS is enabled, both IFrame and JSONP based transports are disabled. As a consequence, IE6 through IE9 are not supported when this mode is enabled.

Allow all origins: to enable this mode, you should provide \* as the allowed origin value. In this mode, all transports are available.

WebSocket and SockJS allowed origins can be configured as shown bellow:

import org.springframework.web.socket.config.annotation.EnableWebSocket;

import org.springframework.web.socket.config.annotation.WebSocketConfigurer;

import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

@Configuration

@EnableWebSocket

public class WebSocketConfig implements WebSocketConfigurer {

@Override

public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {

registry.addHandler(myHandler(), "/myHandler").setAllowedOrigins("http://mydomain.com");

}

@Bean

public WebSocketHandler myHandler() {

return new MyHandler();

}

}

XML configuration equivalent:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:handlers allowed-origins="http://mydomain.com">

<websocket:mapping path="/myHandler" handler="myHandler" />

</websocket:handlers>

<bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>

4.3. SockJS Fallback

Over the public Internet, restrictive proxies outside your control may preclude WebSocket interactions either because they are not configured to pass on the Upgrade header or because they close long lived connections that appear idle.

The solution to this problem is WebSocket emulation, i.e. attempting to use WebSocket first and then falling back on HTTP-based techniques that emulate a WebSocket interaction and expose the same application-level API.

On the Servlet stack the Spring Framework provides both server (and also client) support for the SockJS protocol.

4.3.1. Overview

The goal of SockJS is to let applications use a WebSocket API but fall back to non-WebSocket alternatives when necessary at runtime, i.e. without the need to change application code.

SockJS consists of:

The SockJS protocol defined in the form of executable narrated tests.

The SockJS JavaScript client - a client library for use in browsers.

SockJS server implementations including one in the Spring Framework spring-websocket module.

As of 4.1 spring-websocket also provides a SockJS Java client.

SockJS is designed for use in browsers. It goes to great lengths to support a wide range of browser versions using a variety of techniques. For the full list of SockJS transport types and browsers see the SockJS client page. Transports fall in 3 general categories: WebSocket, HTTP Streaming, and HTTP Long Polling. For an overview of these categories see this blog post.

The SockJS client begins by sending "GET /info" to obtain basic information from the server. After that it must decide what transport to use. If possible WebSocket is used. If not, in most browsers there is at least one HTTP streaming option and if not then HTTP (long) polling is used.

All transport requests have the following URL structure:

http://host:port/myApp/myEndpoint/{server-id}/{session-id}/{transport}

{server-id} - useful for routing requests in a cluster but not used otherwise.

{session-id} - correlates HTTP requests belonging to a SockJS session.

{transport} - indicates the transport type, e.g. "websocket", "xhr-streaming", etc.

The WebSocket transport needs only a single HTTP request to do the WebSocket handshake. All messages thereafter are exchanged on that socket.

HTTP transports require more requests. Ajax/XHR streaming for example relies on one long-running request for server-to-client messages and additional HTTP POST requests for client-to-server messages. Long polling is similar except it ends the current request after each server-to-client send.

SockJS adds minimal message framing. For example the server sends the letter o ("open" frame) initially, messages are sent as a["message1","message2"] (JSON-encoded array), the letter h ("heartbeat" frame) if no messages flow for 25 seconds by default, and the letter c ("close" frame) to close the session.

To learn more, run an example in a browser and watch the HTTP requests. The SockJS client allows fixing the list of transports so it is possible to see each transport one at a time. The SockJS client also provides a debug flag which enables helpful messages in the browser console. On the server side enable TRACE logging for org.springframework.web.socket. For even more detail refer to the SockJS protocol narrated test.

4.3.2. Enable SockJS

SockJS is easy to enable through Java configuration:

@Configuration

@EnableWebSocket

public class WebSocketConfig implements WebSocketConfigurer {

@Override

public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {

registry.addHandler(myHandler(), "/myHandler").withSockJS();

}

@Bean

public WebSocketHandler myHandler() {

return new MyHandler();

}

}

and the XML configuration equivalent:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:handlers>

<websocket:mapping path="/myHandler" handler="myHandler"/>

<websocket:sockjs/>

</websocket:handlers>

<bean id="myHandler" class="org.springframework.samples.MyHandler"/>

</beans>

The above is for use in Spring MVC applications and should be included in the configuration of a DispatcherServlet. However, Spring’s WebSocket and SockJS support does not depend on Spring MVC. It is relatively simple to integrate into other HTTP serving environments with the help of SockJsHttpRequestHandler.

On the browser side, applications can use the sockjs-client (version 1.0.x) that emulates the W3C WebSocket API and communicates with the server to select the best transport option depending on the browser it’s running in. Review the sockjs-client page and the list of transport types supported by browser. The client also provides several configuration options, for example, to specify which transports to include.

4.3.3. IE 8, 9

Internet Explorer 8 and 9 are and will remain common for some time. They are a key reason for having SockJS. This section covers important considerations about running in those browsers.

The SockJS client supports Ajax/XHR streaming in IE 8 and 9 via Microsoft’s XDomainRequest. That works across domains but does not support sending cookies. Cookies are very often essential for Java applications. However since the SockJS client can be used with many server types (not just Java ones), it needs to know whether cookies matter. If so the SockJS client prefers Ajax/XHR for streaming or otherwise it relies on a iframe-based technique.

The very first "/info" request from the SockJS client is a request for information that can influence the client’s choice of transports. One of those details is whether the server application relies on cookies, e.g. for authentication purposes or clustering with sticky sessions. Spring’s SockJS support includes a property called sessionCookieNeeded. It is enabled by default since most Java applications rely on the JSESSIONID cookie. If your application does not need it, you can turn off this option and the SockJS client should choose xdr-streaming in IE 8 and 9.

If you do use an iframe-based transport, and in any case, it is good to know that browsers can be instructed to block the use of IFrames on a given page by setting the HTTP response header X-Frame-Options to DENY, SAMEORIGIN, or ALLOW-FROM <origin>. This is used to prevent clickjacking.

Spring Security 3.2+ provides support for setting X-Frame-Options on every response. By default the Spring Security Java config sets it to DENY. In 3.2 the Spring Security XML namespace does not set that header by default but may be configured to do so, and in the future it may set it by default.

See Section 7.1. "Default Security Headers" of the Spring Security documentation for details on how to configure the setting of the X-Frame-Options header. You may also check or watch SEC-2501 for additional background.

If your application adds the X-Frame-Options response header (as it should!) and relies on an iframe-based transport, you will need to set the header value to SAMEORIGIN or ALLOW-FROM <origin>. Along with that the Spring SockJS support also needs to know the location of the SockJS client because it is loaded from the iframe. By default the iframe is set to download the SockJS client from a CDN location. It is a good idea to configure this option to a URL from the same origin as the application.

In Java config this can be done as shown below. The XML namespace provides a similar option via the <websocket:sockjs> element:

@Configuration

@EnableWebSocket

public class WebSocketConfig implements WebSocketConfigurer {

@Override

public void registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/portfolio").withSockJS()

.setClientLibraryUrl("http://localhost:8080/myapp/js/sockjs-client.js");

}

// ...

}

During initial development, do enable the SockJS client devel mode that prevents the browser from caching SockJS requests (like the iframe) that would otherwise be cached. For details on how to enable it see the SockJS client page.

4.3.4. Heartbeats

The SockJS protocol requires servers to send heartbeat messages to preclude proxies from concluding a connection is hung. The Spring SockJS configuration has a property called heartbeatTime that can be used to customize the frequency. By default a heartbeat is sent after 25 seconds assuming no other messages were sent on that connection. This 25 seconds value is in line with the following IETF recommendation for public Internet applications.

When using STOMP over WebSocket/SockJS, if the STOMP client and server negotiate heartbeats to be exchanged, the SockJS heartbeats are disabled.

The Spring SockJS support also allows configuring the TaskScheduler to use for scheduling heartbeats tasks. The task scheduler is backed by a thread pool with default settings based on the number of available processors. Applications should consider customizing the settings according to their specific needs.

4.3.5. Client disconnects

HTTP streaming and HTTP long polling SockJS transports require a connection to remain open longer than usual. For an overview of these techniques see this blog post.

In Servlet containers this is done through Servlet 3 async support that allows exiting the Servlet container thread processing a request and continuing to write to the response from another thread.

A specific issue is that the Servlet API does not provide notifications for a client that has gone away, see SERVLET\_SPEC-44. However, Servlet containers raise an exception on subsequent attempts to write to the response. Since Spring’s SockJS Service supports sever-sent heartbeats (every 25 seconds by default), that means a client disconnect is usually detected within that time period or earlier if messages are sent more frequently.

As a result network IO failures may occur simply because a client has disconnected, which can fill the log with unnecessary stack traces. Spring makes a best effort to identify such network failures that represent client disconnects (specific to each server) and log a minimal message using the dedicated log category DISCONNECTED\_CLIENT\_LOG\_CATEGORY defined in AbstractSockJsSession. If you need to see the stack traces, set that log category to TRACE.

4.3.6. SockJS and CORS

If you allow cross-origin requests (see Allowed origins), the SockJS protocol uses CORS for cross-domain support in the XHR streaming and polling transports. Therefore CORS headers are added automatically unless the presence of CORS headers in the response is detected. So if an application is already configured to provide CORS support, e.g. through a Servlet Filter, Spring’s SockJsService will skip this part.

It is also possible to disable the addition of these CORS headers via the suppressCors property in Spring’s SockJsService.

The following is the list of headers and values expected by SockJS:

"Access-Control-Allow-Origin" - initialized from the value of the "Origin" request header.

"Access-Control-Allow-Credentials" - always set to true.

"Access-Control-Request-Headers" - initialized from values from the equivalent request header.

"Access-Control-Allow-Methods" - the HTTP methods a transport supports (see TransportType enum).

"Access-Control-Max-Age" - set to 31536000 (1 year).

For the exact implementation see addCorsHeaders in AbstractSockJsService as well as the TransportType enum in the source code.

Alternatively if the CORS configuration allows it consider excluding URLs with the SockJS endpoint prefix thus letting Spring’s SockJsService handle it.

4.3.7. SockJsClient

A SockJS Java client is provided in order to connect to remote SockJS endpoints without using a browser. This can be especially useful when there is a need for bidirectional communication between 2 servers over a public network, i.e. where network proxies may preclude the use of the WebSocket protocol. A SockJS Java client is also very useful for testing purposes, for example to simulate a large number of concurrent users.

The SockJS Java client supports the "websocket", "xhr-streaming", and "xhr-polling" transports. The remaining ones only make sense for use in a browser.

The WebSocketTransport can be configured with:

StandardWebSocketClient in a JSR-356 runtime

JettyWebSocketClient using the Jetty 9+ native WebSocket API

Any implementation of Spring’s WebSocketClient

An XhrTransport by definition supports both "xhr-streaming" and "xhr-polling" since from a client perspective there is no difference other than in the URL used to connect to the server. At present there are two implementations:

RestTemplateXhrTransport uses Spring’s RestTemplate for HTTP requests.

JettyXhrTransport uses Jetty’s HttpClient for HTTP requests.

The example below shows how to create a SockJS client and connect to a SockJS endpoint:

List<Transport> transports = new ArrayList<>(2);

transports.add(new WebSocketTransport(new StandardWebSocketClient()));

transports.add(new RestTemplateXhrTransport());

SockJsClient sockJsClient = new SockJsClient(transports);

sockJsClient.doHandshake(new MyWebSocketHandler(), "ws://example.com:8080/sockjs");

SockJS uses JSON formatted arrays for messages. By default Jackson 2 is used and needs to be on the classpath. Alternatively you can configure a custom implementation of SockJsMessageCodec and configure it on the SockJsClient.

To use the SockJsClient for simulating a large number of concurrent users you will need to configure the underlying HTTP client (for XHR transports) to allow a sufficient number of connections and threads. For example with Jetty:

HttpClient jettyHttpClient = new HttpClient();

jettyHttpClient.setMaxConnectionsPerDestination(1000);

jettyHttpClient.setExecutor(new QueuedThreadPool(1000));

Consider also customizing these server-side SockJS related properties (see Javadoc for details):

@Configuration

public class WebSocketConfig extends WebSocketMessageBrokerConfigurationSupport {

@Override

public void registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/sockjs").withSockJS()

.setStreamBytesLimit(512 \* 1024)

.setHttpMessageCacheSize(1000)

.setDisconnectDelay(30 \* 1000);

}

// ...

}

4.4. STOMP

The WebSocket protocol defines two types of messages, text and binary, but their content is undefined. The defines a mechanism for client and server to negotiate a sub-protocol — i.e. a higher level messaging protocol, to use on top of WebSocket to define what kind of messages each can send, what is the format and content for each message, and so on. The use of a sub-protocol is optional but either way client and server will need to agree on some protocol that defines message content.

4.4.1. Overview

STOMP is a simple, text-oriented messaging protocol that was originally created for scripting languages such as Ruby, Python, and Perl to connect to enterprise message brokers. It is designed to address a minimal subset of commonly used messaging patterns. STOMP can be used over any reliable, 2-way streaming network protocol such as TCP and WebSocket. Although STOMP is a text-oriented protocol, message payloads can be either text or binary.

STOMP is a frame based protocol whose frames are modeled on HTTP. The structure of a STOMP frame:

COMMAND

header1:value1

header2:value2

Body^@

Clients can use the SEND or SUBSCRIBE commands to send or subscribe for messages along with a "destination" header that describes what the message is about and who should receive it. This enables a simple publish-subscribe mechanism that can be used to send messages through the broker to other connected clients or to send messages to the server to request that some work be performed.

When using Spring’s STOMP support, the Spring WebSocket application acts as the STOMP broker to clients. Messages are routed to @Controller message-handling methods or to a simple, in-memory broker that keeps track of subscriptions and broadcasts messages to subscribed users. You can also configure Spring to work with a dedicated STOMP broker (e.g. RabbitMQ, ActiveMQ, etc) for the actual broadcasting of messages. In that case Spring maintains TCP connections to the broker, relays messages to it, and also passes messages from it down to connected WebSocket clients. Thus Spring web applications can rely on unified HTTP-based security, common validation, and a familiar programming model message-handling work.

Here is an example of a client subscribing to receive stock quotes which the server may emit periodically e.g. via a scheduled task sending messages through a SimpMessagingTemplate to the broker:

SUBSCRIBE

id:sub-1

destination:/topic/price.stock.\*

^@

Here is an example of a client sending a trade request, which the server may handle through an @MessageMapping method and later on, after the execution, broadcast a trade confirmation message and details down to the client:

SEND

destination:/queue/trade

content-type:application/json

content-length:44

{"action":"BUY","ticker":"MMM","shares",44}^@

The meaning of a destination is intentionally left opaque in the STOMP spec. It can be any string, and it’s entirely up to STOMP servers to define the semantics and the syntax of the destinations that they support. It is very common, however, for destinations to be path-like strings where "/topic/.." implies publish-subscribe (one-to-many) and "/queue/" implies point-to-point (one-to-one) message exchanges.

STOMP servers can use the MESSAGE command to broadcast messages to all subscribers. Here is an example of a server sending a stock quote to a subscribed client:

MESSAGE

message-id:nxahklf6-1

subscription:sub-1

destination:/topic/price.stock.MMM

{"ticker":"MMM","price":129.45}^@

It is important to know that a server cannot send unsolicited messages. All messages from a server must be in response to a specific client subscription, and the "subscription-id" header of the server message must match the "id" header of the client subscription.

The above overview is intended to provide the most basic understanding of the STOMP protocol. It is recommended to review the protocol specification in full.

4.4.2. Benefits

Use of STOMP as a sub-protocol enables the Spring Framework and Spring Security to provide a richer programming model vs using raw WebSockets. The same point can be made about how HTTP vs raw TCP and how it enables Spring MVC and other web frameworks to provide rich functionality. The following is a list of benefits:

No need to invent a custom messaging protocol and message format.

STOMP clients are available including a Java client in the Spring Framework.

Message brokers such as RabbitMQ, ActiveMQ, and others can be used (optionally) to manage subscriptions and broadcast messages.

Application logic can be organized in any number of @Controller's and messages routed to them based on the STOMP destination header vs handling raw WebSocket messages with a single WebSocketHandler for a given connection.

Use Spring Security to secure messages based on STOMP destinations and message types.

4.4.3. Enable STOMP

STOMP over WebSocket support is available in the spring-messaging and the spring-websocket modules. Once you have those dependencies, you can expose a STOMP endpoints, over WebSocket with SockJS Fallback, as shown below:

import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;

import org.springframework.web.socket.config.annotation.StompEndpointRegistry;

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

@Override

public void registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/portfolio").withSockJS();

}

@Override

public void configureMessageBroker(MessageBrokerRegistry config) {

config.setApplicationDestinationPrefixes("/app");

config.enableSimpleBroker("/topic", "/queue");

}

}

"/portfolio" is the HTTP URL for the endpoint to which a WebSocket (or SockJS) client will need to connect to for the WebSocket handshake.

STOMP messages whose destination header begins with "/app" are routed to @MessageMapping methods in @Controller classes.

Use the built-in, message broker for subscriptions and broadcasting; Route messages whose destination header begins with "/topic" or "/queue" to the broker.

The same configuration in XML:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:message-broker application-destination-prefix="/app">

<websocket:stomp-endpoint path="/portfolio">

<websocket:sockjs/>

</websocket:stomp-endpoint>

<websocket:simple-broker prefix="/topic, /queue"/>

</websocket:message-broker>

</beans>

For the built-in, simple broker the "/topic" and "/queue" prefixes do not have any special meaning. They’re merely a convention to differentiate between pub-sub vs point-to-point messaging (i.e. many subscribers vs one consumer). When using an external broker, please check the STOMP page of the broker to understand what kind of STOMP destinations and prefixes it supports.

To connect from a browser, for SockJS you can use the sockjs-client. For STOMP many applications have used the jmesnil/stomp-websocket library (also known as stomp.js) which is feature complete and has been used in production for years but is no longer maintained. At present the JSteunou/webstomp-client is the most actively maintained and evolving successor of that library and the example code below is based on it:

var socket = new SockJS("/spring-websocket-portfolio/portfolio");

var stompClient = webstomp.over(socket);

stompClient.connect({}, function(frame) {

}

Or if connecting via WebSocket (without SockJS):

var socket = new WebSocket("/spring-websocket-portfolio/portfolio");

var stompClient = Stomp.over(socket);

stompClient.connect({}, function(frame) {

}

Note that the stompClient above does not need to specify login and passcode headers. Even if it did, they would be ignored, or rather overridden, on the server side. See the sections Connect to Broker and Authentication for more information on authentication.

For a more example code see:

Using WebSocket to build an interactive web application getting started guide.

Stock Portfolio sample application.

4.4.4. Flow of Messages

Once a STOMP endpoint is exposed, the Spring application becomes a STOMP broker for connected clients. This section describes the flow of messages on the server side.

The spring-messaging module contains foundational support for messaging applications that originated in Spring Integration and was later extracted and incorporated into the Spring Framework for broader use across many Spring projects and application scenarios. Below is a list of a few of the available messaging abstractions:

Message — simple representation for a message including headers and payload.

MessageHandler — contract for handling a message.

MessageChannel — contract for sending a message that enables loose coupling between producers and consumers.

SubscribableChannel — MessageChannel with MessageHandler subscribers.

ExecutorSubscribableChannel — SubscribableChannel that uses an Executor for delivering messages.

Both the Java config (i.e. @EnableWebSocketMessageBroker) and the XML namespace config (i.e. <websocket:message-broker>) use the above components to assemble a message workflow. The diagram below shows the components used when the simple, built-in message broker is enabled:

message flow simple broker

There are 3 message channels in the above diagram:

"clientInboundChannel" — for passing messages received from WebSocket clients.

"clientOutboundChannel" — for sending server messages to WebSocket clients.

"brokerChannel" — for sending messages to the message broker from within server-side, application code.

The next diagram shows the components used when an external broker (e.g. RabbitMQ) is configured for managing subscriptions and broadcasting messages:

message flow broker relay

The main difference in the above diagram is the use of the "broker relay" for passing messages up to the external STOMP broker over TCP, and for passing messages down from the broker to subscribed clients.

When messages are received from a WebSocket connectin, they’re decoded to STOMP frames, then turned into a Spring Message representation, and sent to the "clientInboundChannel" for further processing. For example STOMP messages whose destination header starts with "/app" may be routed to @MessageMapping methods in annotated controllers, while "/topic" and "/queue" messages may be routed directly to the message broker.

An annotated @Controller handling a STOMP message from a client may send a message to the message broker through the "brokerChannel", and the broker will broadcast the message to matching subscribers through the "clientOutboundChannel". The same controller can also do the same in response to HTTP requests, so a client may perform an HTTP POST and then an @PostMapping method can send a message to the message broker to broadcast to subscribed clients.

Let’s trace the flow through a simple example. Given the following server setup:

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

@Override

public void registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/portfolio");

}

@Override

public void configureMessageBroker(MessageBrokerRegistry registry) {

registry.setApplicationDestinationPrefixes("/app");

registry.enableSimpleBroker("/topic");

}

}

@Controller

public class GreetingController {

@MessageMapping("/greeting") {

public String handle(String greeting) {

return "[" + getTimestamp() + ": " + greeting;

}

}

Client connects to "http://localhost:8080/portfolio" and once a WebSocket connection is established, STOMP frames begin to flow on it.

Client sends SUBSCRIBE frame with destination header "/topic/greeting". Once received and decoded, the message is sent to the "clientInboundChannel", then routed to the message broker which stores the client subscription.

Client sends SEND frame to "/app/greeting". The "/app" prefix helps to route it to annotated controllers. After the "/app" prefix is stripped, the remaining "/greeting" part of the destination is mapped to the @MessageMapping method in GreetingController.

The value returned from GreetingController is turned into a Spring Message with a payload based on the return value and a default destination header of "/topic/greeting" (derived from the input destination with "/app" replaced by "/topic"). The resulting message is sent to the "brokerChannel" and handled by the message broker.

The message broker finds all matching subscribers, and sends a MESSAGE frame to each through the "clientOutboundChannel" from where messages are encoded as STOMP frames and sent on the WebSocket connection.

The next section provides more details on annotated methods including the kinds of arguments and return values supported.

4.4.5. Annotated Controllers

Applications can use annotated @Controller classes to handle messages from clients. Such classes can declare @MessageMapping, @SubscribeMapping, and @ExceptionHandler methods as described next.

@MessageMapping

The @MessageMapping annotation can be used on methods to route messages based on their destination. It is supported at the method level as well as at the type level. At type level @MessageMapping is used to express shared mappings across all methods in a controller.

By default destination mappings are expected to be Ant-style, path patterns, e.g. "/foo\*", "/foo/\*\*". The patterns include support for template variables, e.g. "/foo/{id}", that can be referenced with @DestinationVariable method arguments.

Applications can choose to switch to a dot-separated destination convention. See Dot as Separator.

@MessageMapping methods can have flexible signatures with the following arguments:

Method argument Description

Message

For access to the complete message.

MessageHeaders

For access to the headers within the Message.

MessageHeaderAccessor, SimpMessageHeaderAccessor, StompHeaderAccessor

For access to the headers via typed accessor methods.

@Payload

For access to the payload of the message, converted (e.g. from JSON) via a configured MessageConverter.

The presence of this annotation is not required since it is assumed by default if no other argument is matched.

Payload arguments may be annotated with @javax.validation.Valid or Spring’s @Validated in order to be automatically validated.

@Header

For access to a specific header value along with type conversion using an org.springframework.core.convert.converter.Converter if necessary.

@Headers

For access to all headers in the message. This argument must be assignable to java.util.Map.

@DestinationVariable

For access to template variables extracted from the message destination. Values will be converted to the declared method argument type as necessary.

java.security.Principal

Reflects the user logged in at the time of the WebSocket HTTP handshake.

When an @MessageMapping method returns a value, by default the value is serialized to a payload through a configured MessageConverter, and then sent as a Message to the "brokerChannel" from where it is broadcast to subscribers. The destination of the outbound message is the same as that of the inbound message but prefixed with "/topic".

You can use the @SendTo method annotation to customize the destination to send the payload to. @SendTo can also be used at the class level to share a default target destination to send messages to. @SendToUser is an variant for sending messages only to the user associated with a message. See User Destinations for details.

The return value from an @MessageMapping method may be wrapped with ListenableFuture, CompletableFuture, or CompletionStage in order to produce the payload asynchronously.

As an alternative to returning a payload from an @MessageMapping method you can also send messages using the SimpMessagingTemplate, which is also how return values are handled under the covers. See Send Messages.

@SubscribeMapping

@SubscribeMapping is similar to @MessageMapping but narrows the mapping to subscription messages only. It supports the same method arguments as @MessageMapping does. However for the return value, by default a message is sent directly to the client via "clientOutboundChannel" in response to the subscription, and not to the broker via "brokerChannel" as a broadcast to matching subscriptions. Adding @SendTo or @SendToUser overrides this behavior and sends to the broker instead.

When is this useful? Let’s assume the broker is mapped to "/topic" and "/queue" while application controllers are mapped to "/app". In this setup, the broker stores all subscriptions to "/topic" and "/queue" that are intended for repeated broadcasts, and there is no need for the application to get involved. A client could also also subscribe to some "/app" destination and a controller could return a value in response to that subscription without involving the broker, effectively a one-off, request-reply exchange, without storing or using the subscription again. One case for this is populating a UI with initial data on startup.

When is this not useful? Do not try to map broker and controllers to the same destination prefix unless you want both to process messages, including subscriptions, independently for some reason. Inbound messages are handled in parallel. There are no guarantees whether broker or controller will process a given message first. If the goal is to be notified when a subscription is stored and ready for broadcasts, then a client should ask for a receipt if the server supports it (simple broker does not). For example with the Java STOMP Client:

@Autowired

private TaskScheduler messageBrokerTaskScheduler;

// During initialization..

stompClient.setTaskScheduler(this.messageBrokerTaskScheduler);

// When subscribing..

StompHeaders headers = new StompHeaders();

headers.setDestination("/topic/...");

headers.setReceipt("r1");

FrameHandler handler = ...;

stompSession.subscribe(headers, handler).addReceiptTask(() -> {

// Subscription ready...

});

A server side option is to register an ExecutorChannelInterceptor on the brokerChannel and implement the afterMessageHandled method that is invoked after messages, including subscriptions, have been handled.

@MessageExceptionHandler

An application can use @MessageExceptionHandler methods to handle exceptions from @MessageMapping methods. Exceptions of interest can be declared in the annotation itself, or through a method argument if you want to get access to the exception instance:

@Controller

public class MyController {

// ...

@MessageExceptionHandler

public ApplicationError handleException(MyException exception) {

// ...

return appError;

}

}

@MessageExceptionHandler methods support flexible method signatures and support the same method argument types and return values as @MessageMapping methods.

Typically @MessageExceptionHandler methods apply within the @Controller class (or class hierarchy) they are declared in. If you want such methods to apply more globally, across controllers, you can declare them in a class marked with @ControllerAdvice. This is comparable to similar support in Spring MVC.

4.4.6. Send Messages

What if you want to send messages to connected clients from any part of the application? Any application component can send messages to the "brokerChannel". The easiest way to do that is to have a SimpMessagingTemplate injected, and use it to send messages. Typically it should be easy to have it injected by type, for example:

@Controller

public class GreetingController {

private SimpMessagingTemplate template;

@Autowired

public GreetingController(SimpMessagingTemplate template) {

this.template = template;

}

@RequestMapping(path="/greetings", method=POST)

public void greet(String greeting) {

String text = "[" + getTimestamp() + "]:" + greeting;

this.template.convertAndSend("/topic/greetings", text);

}

}

But it can also be qualified by its name "brokerMessagingTemplate" if another bean of the same type exists.

4.4.7. Simple Broker

The built-in, simple message broker handles subscription requests from clients, stores them in memory, and broadcasts messages to connected clients with matching destinations. The broker supports path-like destinations, including subscriptions to Ant-style destination patterns.

Applications can also use dot-separated destinations (vs slash). See Dot as Separator.

4.4.8. External Broker

The simple broker is great for getting started but supports only a subset of STOMP commands (e.g. no acks, receipts, etc.), relies on a simple message sending loop, and is not suitable for clustering. As an alternative, applications can upgrade to using a full-featured message broker.

Check the STOMP documentation for your message broker of choice (e.g. RabbitMQ, ActiveMQ, etc.), install the broker, and run it with STOMP support enabled. Then enable the STOMP broker relay in the Spring configuration instead of the simple broker.

Below is example configuration that enables a full-featured broker:

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

@Override

public void registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/portfolio").withSockJS();

}

@Override

public void configureMessageBroker(MessageBrokerRegistry registry) {

registry.enableStompBrokerRelay("/topic", "/queue");

registry.setApplicationDestinationPrefixes("/app");

}

}

XML configuration equivalent:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:message-broker application-destination-prefix="/app">

<websocket:stomp-endpoint path="/portfolio" />

<websocket:sockjs/>

</websocket:stomp-endpoint>

<websocket:stomp-broker-relay prefix="/topic,/queue" />

</websocket:message-broker>

</beans>

The "STOMP broker relay" in the above configuration is a Spring MessageHandler that handles messages by forwarding them to an external message broker. To do so it establishes TCP connections to the broker, forwards all messages to it, and then forwards all messages received from the broker to clients through their WebSocket sessions. Essentially it acts as a "relay" that forwards messages in both directions.

Please add io.projectreactor.ipc:reactor-netty and io.netty:netty-all dependencies to your project for TCP connection management.

Furthermore, application components (e.g. HTTP request handling methods, business services, etc.) can also send messages to the broker relay, as described in Send Messages, in order to broadcast messages to subscribed WebSocket clients.

In effect, the broker relay enables robust and scalable message broadcasting.

4.4.9. Connect to Broker

A STOMP broker relay maintains a single "system" TCP connection to the broker. This connection is used for messages originating from the server-side application only, not for receiving messages. You can configure the STOMP credentials for this connection, i.e. the STOMP frame login and passcode headers. This is exposed in both the XML namespace and the Java config as the systemLogin/systemPasscode properties with default values guest/guest.

The STOMP broker relay also creates a separate TCP connection for every connected WebSocket client. You can configure the STOMP credentials to use for all TCP connections created on behalf of clients. This is exposed in both the XML namespace and the Java config as the clientLogin/clientPasscode properties with default values guest/guest.

The STOMP broker relay always sets the login and passcode headers on every CONNECT frame that it forwards to the broker on behalf of clients. Therefore WebSocket clients need not set those headers; they will be ignored. As the Authentication section explains, instead WebSocket clients should rely on HTTP authentication to protect the WebSocket endpoint and establish the client identity.

The STOMP broker relay also sends and receives heartbeats to and from the message broker over the "system" TCP connection. You can configure the intervals for sending and receiving heartbeats (10 seconds each by default). If connectivity to the broker is lost, the broker relay will continue to try to reconnect, every 5 seconds, until it succeeds.

Any Spring bean can implement ApplicationListener<BrokerAvailabilityEvent> in order to receive notifications when the "system" connection to the broker is lost and re-established. For example a Stock Quote service broadcasting stock quotes can stop trying to send messages when there is no active "system" connection.

By default, the STOMP broker relay always connects, and reconnects as needed if connectivity is lost, to the same host and port. If you wish to supply multiple addresses, on each attempt to connect, you can configure a supplier of addresses, instead of a fixed host and port. For example:

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

// ...

@Override

public void configureMessageBroker(MessageBrokerRegistry registry) {

registry.enableStompBrokerRelay("/queue/", "/topic/").setTcpClient(createTcpClient());

registry.setApplicationDestinationPrefixes("/app");

}

private ReactorNettyTcpClient<byte[]> createTcpClient() {

Consumer<ClientOptions.Builder<?>> builderConsumer = builder -> {

builder.connectAddress(()-> {

// Select address to connect to ...

});

};

return new ReactorNettyTcpClient<>(builderConsumer, new StompReactorNettyCodec());

}

}

The STOMP broker relay can also be configured with a virtualHost property. The value of this property will be set as the host header of every CONNECT frame and may be useful for example in a cloud environment where the actual host to which the TCP connection is established is different from the host providing the cloud-based STOMP service.

4.4.10. Dot as Separator

When messages are routed to @MessageMapping methods, they’re matched with AntPathMatcher and by default patterns are expected to use slash "/" as separator. This is a good convention in a web applications and similar to HTTP URLs. However if you are more used to messaging conventions, you can switch to using dot "." as separator.

In Java config:

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

// ...

@Override

public void configureMessageBroker(MessageBrokerRegistry registry) {

registry.setPathMatcher(new AntPathMatcher("."));

registry.enableStompBrokerRelay("/queue", "/topic");

registry.setApplicationDestinationPrefixes("/app");

}

}

In XML:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:message-broker application-destination-prefix="/app" path-matcher="pathMatcher">

<websocket:stomp-endpoint path="/stomp"/>

<websocket:stomp-broker-relay prefix="/topic,/queue" />

</websocket:message-broker>

<bean id="pathMatcher" class="org.springframework.util.AntPathMatcher">

<constructor-arg index="0" value="."/>

</bean>

</beans>

After that a controller may use dot "." as separator in @MessageMapping methods:

@Controller

@MessageMapping("foo")

public class FooController {

@MessageMapping("bar.{baz}")

public void handleBaz(@DestinationVariable String baz) {

// ...

}

}

The client can now send a message to "/app/foo.bar.baz123".

In the example above we did not change the prefixes on the "broker relay" because those depend entirely on the external message broker. Check the STOMP documentation pages of the broker you’re using to see what conventions it supports for the destination header.

The "simple broker" on the other hand does rely on the configured PathMatcher so if you switch the separator that will also apply to the broker and the way matches destinations from a message to patterns in subscriptions.

4.4.11. Authentication

Every STOMP over WebSocket messaging session begins with an HTTP request — that can be a request to upgrade to WebSockets (i.e. a WebSocket handshake) or in the case of SockJS fallbacks a series of SockJS HTTP transport requests.

Web applications already have authentication and authorization in place to secure HTTP requests. Typically a user is authenticated via Spring Security using some mechanism such as a login page, HTTP basic authentication, or other. The security context for the authenticated user is saved in the HTTP session and is associated with subsequent requests in the same cookie-based session.

Therefore for a WebSocket handshake, or for SockJS HTTP transport requests, typically there will already be an authenticated user accessible via HttpServletRequest#getUserPrincipal(). Spring automatically associates that user with a WebSocket or SockJS session created for them and subsequently with all STOMP messages transported over that session through a user header.

In short there is nothing special a typical web application needs to do above and beyond what it already does for security. The user is authenticated at the HTTP request level with a security context maintained through a cookie-based HTTP session which is then associated with WebSocket or SockJS sessions created for that user and results in a user header stamped on every Message flowing through the application.

Note that the STOMP protocol does have a "login" and "passcode" headers on the CONNECT frame. Those were originally designed for and are still needed for example for STOMP over TCP. However for STOMP over WebSocket by default Spring ignores authorization headers at the STOMP protocol level and assumes the user is already authenticated at the HTTP transport level and expects that the WebSocket or SockJS session contain the authenticated user.

Spring Security provides WebSocket sub-protocol authorization that uses a ChannelInterceptor to authorize messages based on the user header in them. Also Spring Session provides a WebSocket integration that ensures the user HTTP session does not expire when the WebSocket session is still active.

4.4.12. Token Authentication

Spring Security OAuth provides support for token based security including JSON Web Token (JWT). This can be used as the authentication mechanism in Web applications including STOMP over WebSocket interactions just as described in the previous section, i.e. maintaining identity through a cookie-based session.

At the same time cookie-based sessions are not always the best fit for example in applications that don’t wish to maintain a server-side session at all or in mobile applications where it’s common to use headers for authentication.

The WebSocket protocol RFC 6455 "doesn’t prescribe any particular way that servers can authenticate clients during the WebSocket handshake." In practice however browser clients can only use standard authentication headers (i.e. basic HTTP authentication) or cookies and cannot for example provide custom headers. Likewise the SockJS JavaScript client does not provide a way to send HTTP headers with SockJS transport requests, see sockjs-client issue 196. Instead it does allow sending query parameters that can be used to send a token but that has its own drawbacks, for example as the token may be inadvertently logged with the URL in server logs.

The above limitations are for browser-based clients and do not apply to the Spring Java-based STOMP client which does support sending headers with both WebSocket and SockJS requests.

Therefore applications that wish to avoid the use of cookies may not have any good alternatives for authentication at the HTTP protocol level. Instead of using cookies they may prefer to authenticate with headers at the STOMP messaging protocol level There are 2 simple steps to doing that:

Use the STOMP client to pass authentication header(s) at connect time.

Process the authentication header(s) with a ChannelInterceptor.

Below is the example server-side configuration to register a custom authentication interceptor. Note that an interceptor only needs to authenticate and set the user header on the CONNECT Message. Spring will note and save the authenticated user and associate it with subsequent STOMP messages on the same session:

@Configuration

@EnableWebSocketMessageBroker

public class MyConfig implements WebSocketMessageBrokerConfigurer {

@Override

public void configureClientInboundChannel(ChannelRegistration registration) {

registration.setInterceptors(new ChannelInterceptorAdapter() {

@Override

public Message<?> preSend(Message<?> message, MessageChannel channel) {

StompHeaderAccessor accessor =

MessageHeaderAccessor.getAccessor(message, StompHeaderAccessor.class);

if (StompCommand.CONNECT.equals(accessor.getCommand())) {

Authentication user = ... ; // access authentication header(s)

accessor.setUser(user);

}

return message;

}

});

}

}

Also note that when using Spring Security’s authorization for messages, at present you will need to ensure that the authentication ChannelInterceptor config is ordered ahead of Spring Security’s. This is best done by declaring the custom interceptor in its own implementation of WebSocketMessageBrokerConfigurer marked with @Order(Ordered.HIGHEST\_PRECEDENCE + 99).

4.4.13. User Destinations

An application can send messages targeting a specific user, and Spring’s STOMP support recognizes destinations prefixed with "/user/" for this purpose. For example, a client might subscribe to the destination "/user/queue/position-updates". This destination will be handled by the UserDestinationMessageHandler and transformed into a destination unique to the user session, e.g. "/queue/position-updates-user123". This provides the convenience of subscribing to a generically named destination while at the same time ensuring no collisions with other users subscribing to the same destination so that each user can receive unique stock position updates.

On the sending side messages can be sent to a destination such as "/user/{username}/queue/position-updates", which in turn will be translated by the UserDestinationMessageHandler into one or more destinations, one for each session associated with the user. This allows any component within the application to send messages targeting a specific user without necessarily knowing anything more than their name and the generic destination. This is also supported through an annotation as well as a messaging template.

For example, a message-handling method can send messages to the user associated with the message being handled through the @SendToUser annotation (also supported on the class-level to share a common destination):

@Controller

public class PortfolioController {

@MessageMapping("/trade")

@SendToUser("/queue/position-updates")

public TradeResult executeTrade(Trade trade, Principal principal) {

// ...

return tradeResult;

}

}

If the user has more than one session, by default all of the sessions subscribed to the given destination are targeted. However sometimes, it may be necessary to target only the session that sent the message being handled. This can be done by setting the broadcast attribute to false, for example:

@Controller

public class MyController {

@MessageMapping("/action")

public void handleAction() throws Exception{

// raise MyBusinessException here

}

@MessageExceptionHandler

@SendToUser(destinations="/queue/errors", broadcast=false)

public ApplicationError handleException(MyBusinessException exception) {

// ...

return appError;

}

}

While user destinations generally imply an authenticated user, it isn’t required strictly. A WebSocket session that is not associated with an authenticated user can subscribe to a user destination. In such cases the @SendToUser annotation will behave exactly the same as with broadcast=false, i.e. targeting only the session that sent the message being handled.

It is also possible to send a message to user destinations from any application component by injecting the SimpMessagingTemplate created by the Java config or XML namespace, for example (the bean name is "brokerMessagingTemplate" if required for qualification with @Qualifier):

@Service

public class TradeServiceImpl implements TradeService {

private final SimpMessagingTemplate messagingTemplate;

@Autowired

public TradeServiceImpl(SimpMessagingTemplate messagingTemplate) {

this.messagingTemplate = messagingTemplate;

}

// ...

public void afterTradeExecuted(Trade trade) {

this.messagingTemplate.convertAndSendToUser(

trade.getUserName(), "/queue/position-updates", trade.getResult());

}

}

When using user destinations with an external message broker, check the broker documentation on how to manage inactive queues, so that when the user session is over, all unique user queues are removed. For example, RabbitMQ creates auto-delete queues when destinations like /exchange/amq.direct/position-updates are used. So in that case the client could subscribe to /user/exchange/amq.direct/position-updates. Similarly, ActiveMQ has configuration options for purging inactive destinations.

In a multi-application server scenario a user destination may remain unresolved because the user is connected to a different server. In such cases you can configure a destination to broadcast unresolved messages to so that other servers have a chance to try. This can be done through the userDestinationBroadcast property of the MessageBrokerRegistry in Java config and the user-destination-broadcast attribute of the message-broker element in XML.

4.4.14. Order of Messages

Messages from the broker are published to the "clientOutboundChannel" from where they are written to WebSocket sessions. As the channel is backed by a ThreadPoolExecutor messages are processed in different threads, and the resulting sequence received by the client may not match the exact order of publication.

If this is an issue, enable the following flag:

@Configuration

@EnableWebSocketMessageBroker

public class MyConfig implements WebSocketMessageBrokerConfigurer {

@Override

protected void configureMessageBroker(MessageBrokerRegistry registry) {

// ...

registry.setPreservePublishOrder(true);

}

}

The same in XML:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:message-broker preserve-publish-order="true">

<!-- ... -->

</websocket:message-broker>

</beans>

When the flag is set, messages within the same client session are published to the "clientOutboundChannel" one at a time, so that the order of publication is guaranteed. Note that this incurs a small performance overhead, so enable it only if required.

4.4.15. Events

Several ApplicationContext events (listed below) are published and can be received by implementing Spring’s ApplicationListener interface.

BrokerAvailabilityEvent — indicates when the broker becomes available/unavailable. While the "simple" broker becomes available immediately on startup and remains so while the application is running, the STOMP "broker relay" may lose its connection to the full featured broker, for example if the broker is restarted. The broker relay has reconnect logic and will re-establish the "system" connection to the broker when it comes back, hence this event is published whenever the state changes from connected to disconnected and vice versa. Components using the SimpMessagingTemplate should subscribe to this event and avoid sending messages at times when the broker is not available. In any case they should be prepared to handle MessageDeliveryException when sending a message.

SessionConnectEvent — published when a new STOMP CONNECT is received indicating the start of a new client session. The event contains the message representing the connect including the session id, user information (if any), and any custom headers the client may have sent. This is useful for tracking client sessions. Components subscribed to this event can wrap the contained message using SimpMessageHeaderAccessor or StompMessageHeaderAccessor.

SessionConnectedEvent — published shortly after a SessionConnectEvent when the broker has sent a STOMP CONNECTED frame in response to the CONNECT. At this point the STOMP session can be considered fully established.

SessionSubscribeEvent — published when a new STOMP SUBSCRIBE is received.

SessionUnsubscribeEvent — published when a new STOMP UNSUBSCRIBE is received.

SessionDisconnectEvent — published when a STOMP session ends. The DISCONNECT may have been sent from the client, or it may also be automatically generated when the WebSocket session is closed. In some cases this event may be published more than once per session. Components should be idempotent with regard to multiple disconnect events.

When using a full-featured broker, the STOMP "broker relay" automatically reconnects the "system" connection in case the broker becomes temporarily unavailable. Client connections however are not automatically reconnected. Assuming heartbeats are enabled, the client will typically notice the broker is not responding within 10 seconds. Clients need to implement their own reconnect logic.

4.4.16. Interception

Events provide notifications for the lifecycle of a STOMP connection and not for every client message. Applications can also register a ChannelInterceptor to intercept any message, and in any part of the processing chain. For example to intercept inbound messages from clients:

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

@Override

public void configureClientInboundChannel(ChannelRegistration registration) {

registration.setInterceptors(new MyChannelInterceptor());

}

}

A custom ChannelInterceptor can use StompHeaderAccessor or SimpMessageHeaderAccessor to access information about the message.

public class MyChannelInterceptor implements ChannelInterceptor {

@Override

public Message<?> preSend(Message<?> message, MessageChannel channel) {

StompHeaderAccessor accessor = StompHeaderAccessor.wrap(message);

StompCommand command = accessor.getStompCommand();

// ...

return message;

}

}

Applications may also implement ExecutorChannelInterceptor which is a sub-interface of ChannelInterceptor with callbacks in the thread in which the messages are handled. While a ChannelInterceptor is invoked once for per message sent to a channel, the ExecutorChannelInterceptor provides hooks in the thread of each MessageHandler subscribed to messages from the channel.

Note that just like with the SesionDisconnectEvent above, a DISCONNECT message may have been sent from the client, or it may also be automatically generated when the WebSocket session is closed. In some cases an interceptor may intercept this message more than once per session. Components should be idempotent with regard to multiple disconnect events.

4.4.17. STOMP Client

Spring provides a STOMP over WebSocket client and a STOMP over TCP client.

To begin create and configure WebSocketStompClient:

WebSocketClient webSocketClient = new StandardWebSocketClient();

WebSocketStompClient stompClient = new WebSocketStompClient(webSocketClient);

stompClient.setMessageConverter(new StringMessageConverter());

stompClient.setTaskScheduler(taskScheduler); // for heartbeats

In the above example StandardWebSocketClient could be replaced with SockJsClient since that is also an implementation of WebSocketClient. The SockJsClient can use WebSocket or HTTP-based transport as a fallback. For more details see SockJsClient.

Next establish a connection and provide a handler for the STOMP session:

String url = "ws://127.0.0.1:8080/endpoint";

StompSessionHandler sessionHandler = new MyStompSessionHandler();

stompClient.connect(url, sessionHandler);

When the session is ready for use the handler is notified:

public class MyStompSessionHandler extends StompSessionHandlerAdapter {

@Override

public void afterConnected(StompSession session, StompHeaders connectedHeaders) {

// ...

}

}

Once the session is established any payload can be sent and that will be serialized with the configured MessageConverter:

session.send("/topic/foo", "payload");

You can also subscribe to destinations. The subscribe methods require a handler for messages on the subscription and return a Subscription handle that can be used to unsubscribe. For each received message the handler can specify the target Object type the payload should be deserialized to:

session.subscribe("/topic/foo", new StompFrameHandler() {

@Override

public Type getPayloadType(StompHeaders headers) {

return String.class;

}

@Override

public void handleFrame(StompHeaders headers, Object payload) {

// ...

}

});

To enable STOMP heartbeat configure WebSocketStompClient with a TaskScheduler and optionally customize the heartbeat intervals, 10 seconds for write inactivity which causes a heartbeat to be sent and 10 seconds for read inactivity which closes the connection.

When using WebSocketStompClient for performance tests to simulate thousands of clients from the same machine consider turning off heartbeats since each connection schedules its own heartbeat tasks and that’s not optimized for a a large number of clients running on the same machine.

The STOMP protocol also supports receipts where the client must add a "receipt" header to which the server responds with a RECEIPT frame after the send or subscribe are processed. To support this the StompSession offers setAutoReceipt(boolean) that causes a "receipt" header to be added on every subsequent send or subscribe. Alternatively you can also manually add a "receipt" header to the StompHeaders. Both send and subscribe return an instance of Receiptable that can be used to register for receipt success and failure callbacks. For this feature the client must be configured with a TaskScheduler and the amount of time before a receipt expires (15 seconds by default).

Note that StompSessionHandler itself is a StompFrameHandler which allows it to handle ERROR frames in addition to the handleException callback for exceptions from the handling of messages, and handleTransportError for transport-level errors including ConnectionLostException.

4.4.18. WebSocket Scope

Each WebSocket session has a map of attributes. The map is attached as a header to inbound client messages and may be accessed from a controller method, for example:

@Controller

public class MyController {

@MessageMapping("/action")

public void handle(SimpMessageHeaderAccessor headerAccessor) {

Map<String, Object> attrs = headerAccessor.getSessionAttributes();

// ...

}

}

It is also possible to declare a Spring-managed bean in the websocket scope. WebSocket-scoped beans can be injected into controllers and any channel interceptors registered on the "clientInboundChannel". Those are typically singletons and live longer than any individual WebSocket session. Therefore you will need to use a scope proxy mode for WebSocket-scoped beans:

@Component

@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class MyBean {

@PostConstruct

public void init() {

// Invoked after dependencies injected

}

// ...

@PreDestroy

public void destroy() {

// Invoked when the WebSocket session ends

}

}

@Controller

public class MyController {

private final MyBean myBean;

@Autowired

public MyController(MyBean myBean) {

this.myBean = myBean;

}

@MessageMapping("/action")

public void handle() {

// this.myBean from the current WebSocket session

}

}

As with any custom scope, Spring initializes a new MyBean instance the first time it is accessed from the controller and stores the instance in the WebSocket session attributes. The same instance is returned subsequently until the session ends. WebSocket-scoped beans will have all Spring lifecycle methods invoked as shown in the examples above.

4.4.19. Performance

There is no silver bullet when it comes to performance. Many factors may affect it including the size of messages, the volume, whether application methods perform work that requires blocking, as well as external factors such as network speed and others. The goal of this section is to provide an overview of the available configuration options along with some thoughts on how to reason about scaling.

In a messaging application messages are passed through channels for asynchronous executions backed by thread pools. Configuring such an application requires good knowledge of the channels and the flow of messages. Therefore it is recommended to review Flow of Messages.

The obvious place to start is to configure the thread pools backing the "clientInboundChannel" and the "clientOutboundChannel". By default both are configured at twice the number of available processors.

If the handling of messages in annotated methods is mainly CPU bound then the number of threads for the "clientInboundChannel" should remain close to the number of processors. If the work they do is more IO bound and requires blocking or waiting on a database or other external system then the thread pool size will need to be increased.

ThreadPoolExecutor has 3 important properties. Those are the core and the max thread pool size as well as the capacity for the queue to store tasks for which there are no available threads.

A common point of confusion is that configuring the core pool size (e.g. 10) and max pool size (e.g. 20) results in a thread pool with 10 to 20 threads. In fact if the capacity is left at its default value of Integer.MAX\_VALUE then the thread pool will never increase beyond the core pool size since all additional tasks will be queued.

Please review the Javadoc of ThreadPoolExecutor to learn how these properties work and understand the various queuing strategies.

On the "clientOutboundChannel" side it is all about sending messages to WebSocket clients. If clients are on a fast network then the number of threads should remain close to the number of available processors. If they are slow or on low bandwidth they will take longer to consume messages and put a burden on the thread pool. Therefore increasing the thread pool size will be necessary.

While the workload for the "clientInboundChannel" is possible to predict — after all it is based on what the application does — how to configure the "clientOutboundChannel" is harder as it is based on factors beyond the control of the application. For this reason there are two additional properties related to the sending of messages. Those are the "sendTimeLimit" and the "sendBufferSizeLimit". Those are used to configure how long a send is allowed to take and how much data can be buffered when sending messages to a client.

The general idea is that at any given time only a single thread may be used to send to a client. All additional messages meanwhile get buffered and you can use these properties to decide how long sending a message is allowed to take and how much data can be buffered in the mean time. Please review the Javadoc and documentation of the XML schema for this configuration for important additional details.

Here is example configuration:

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

@Override

public void configureWebSocketTransport(WebSocketTransportRegistration registration) {

registration.setSendTimeLimit(15 \* 1000).setSendBufferSizeLimit(512 \* 1024);

}

// ...

}

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:message-broker>

<websocket:transport send-timeout="15000" send-buffer-size="524288" />

<!-- ... -->

</websocket:message-broker>

</beans>

The WebSocket transport configuration shown above can also be used to configure the maximum allowed size for incoming STOMP messages. Although in theory a WebSocket message can be almost unlimited in size, in practice WebSocket servers impose limits — for example, 8K on Tomcat and 64K on Jetty. For this reason STOMP clients such as the JavaScript webstomp-client and others split larger STOMP messages at 16K boundaries and send them as multiple WebSocket messages thus requiring the server to buffer and re-assemble.

Spring’s STOMP over WebSocket support does this so applications can configure the maximum size for STOMP messages irrespective of WebSocket server specific message sizes. Do keep in mind that the WebSocket message size will be automatically adjusted if necessary to ensure they can carry 16K WebSocket messages at a minimum.

Here is example configuration:

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

@Override

public void configureWebSocketTransport(WebSocketTransportRegistration registration) {

registration.setMessageSizeLimit(128 \* 1024);

}

// ...

}

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:websocket="http://www.springframework.org/schema/websocket"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/websocket

http://www.springframework.org/schema/websocket/spring-websocket.xsd">

<websocket:message-broker>

<websocket:transport message-size="131072" />

<!-- ... -->

</websocket:message-broker>

</beans>

An important point about scaling is using multiple application instances. Currently it is not possible to do that with the simple broker. However when using a full-featured broker such as RabbitMQ, each application instance connects to the broker and messages broadcast from one application instance can be broadcast through the broker to WebSocket clients connected through any other application instances.

4.4.20. Monitoring

When using @EnableWebSocketMessageBroker or <websocket:message-broker> key infrastructure components automatically gather stats and counters that provide important insight into the internal state of the application. The configuration also declares a bean of type WebSocketMessageBrokerStats that gathers all available information in one place and by default logs it at INFO level once every 30 minutes. This bean can be exported to JMX through Spring’s MBeanExporter for viewing at runtime, for example through JDK’s jconsole. Below is a summary of the available information.

Client WebSocket Sessions

Current

indicates how many client sessions there are currently with the count further broken down by WebSocket vs HTTP streaming and polling SockJS sessions.

Total

indicates how many total sessions have been established.

Abnormally Closed

Connect Failures

these are sessions that got established but were closed after not having received any messages within 60 seconds. This is usually an indication of proxy or network issues.

Send Limit Exceeded

sessions closed after exceeding the configured send timeout or the send buffer limits which can occur with slow clients (see previous section).

Transport Errors

sessions closed after a transport error such as failure to read or write to a WebSocket connection or HTTP request/response.

STOMP Frames

the total number of CONNECT, CONNECTED, and DISCONNECT frames processed indicating how many clients connected on the STOMP level. Note that the DISCONNECT count may be lower when sessions get closed abnormally or when clients close without sending a DISCONNECT frame.

STOMP Broker Relay

TCP Connections

indicates how many TCP connections on behalf of client WebSocket sessions are established to the broker. This should be equal to the number of client WebSocket sessions + 1 additional shared "system" connection for sending messages from within the application.

STOMP Frames

the total number of CONNECT, CONNECTED, and DISCONNECT frames forwarded to or received from the broker on behalf of clients. Note that a DISCONNECT frame is sent to the broker regardless of how the client WebSocket session was closed. Therefore a lower DISCONNECT frame count is an indication that the broker is pro-actively closing connections, may be because of a heartbeat that didn’t arrive in time, an invalid input frame, or other.

Client Inbound Channel

stats from thread pool backing the "clientInboundChannel" providing insight into the health of incoming message processing. Tasks queueing up here is an indication the application may be too slow to handle messages. If there I/O bound tasks (e.g. slow database query, HTTP request to 3rd party REST API, etc) consider increasing the thread pool size.

Client Outbound Channel

stats from the thread pool backing the "clientOutboundChannel" providing insight into the health of broadcasting messages to clients. Tasks queueing up here is an indication clients are too slow to consume messages. One way to address this is to increase the thread pool size to accommodate the number of concurrent slow clients expected. Another option is to reduce the send timeout and send buffer size limits (see the previous section).

SockJS Task Scheduler

stats from thread pool of the SockJS task scheduler which is used to send heartbeats. Note that when heartbeats are negotiated on the STOMP level the SockJS heartbeats are disabled.

4.4.21. Testing

There are two main approaches to testing applications using Spring’s STOMP over WebSocket support. The first is to write server-side tests verifying the functionality of controllers and their annotated message handling methods. The second is to write full end-to-end tests that involve running a client and a server.

The two approaches are not mutually exclusive. On the contrary each has a place in an overall test strategy. Server-side tests are more focused and easier to write and maintain. End-to-end integration tests on the other hand are more complete and test much more, but they’re also more involved to write and maintain.

The simplest form of server-side tests is to write controller unit tests. However this is not useful enough since much of what a controller does depends on its annotations. Pure unit tests simply can’t test that.

Ideally controllers under test should be invoked as they are at runtime, much like the approach to testing controllers handling HTTP requests using the Spring MVC Test framework. i.e. without running a Servlet container but relying on the Spring Framework to invoke the annotated controllers. Just like with Spring MVC Test here there are two two possible alternatives, either using a "context-based" or "standalone" setup:

Load the actual Spring configuration with the help of the Spring TestContext framework, inject "clientInboundChannel" as a test field, and use it to send messages to be handled by controller methods.

Manually set up the minimum Spring framework infrastructure required to invoke controllers (namely the SimpAnnotationMethodMessageHandler) and pass messages for controllers directly to it.

Both of these setup scenarios are demonstrated in the tests for the stock portfolio sample application.

The second approach is to create end-to-end integration tests. For that you will need to run a WebSocket server in embedded mode and connect to it as a WebSocket client sending WebSocket messages containing STOMP frames. The tests for the stock portfolio sample application also demonstrates this approach using Tomcat as the embedded WebSocket server and a simple STOMP client for test purposes.

5. Other Web Frameworks

5.1. Introduction

This chapter details Spring’s integration with third party web frameworks.

One of the core value propositions of the Spring Framework is that of enabling choice. In a general sense, Spring does not force one to use or buy into any particular architecture, technology, or methodology (although it certainly recommends some over others). This freedom to pick and choose the architecture, technology, or methodology that is most relevant to a developer and their development team is arguably most evident in the web area, where Spring provides its own web framework (Spring MVC), while at the same time providing integration with a number of popular third party web frameworks.

5.2. Common config

Before diving into the integration specifics of each supported web framework, let us first take a look at the Spring configuration that is not specific to any one web framework. (This section is equally applicable to Spring’s own web framework, Spring MVC.)

One of the concepts (for want of a better word) espoused by (Spring’s) lightweight application model is that of a layered architecture. Remember that in a 'classic' layered architecture, the web layer is but one of many layers; it serves as one of the entry points into a server side application and it delegates to service objects (facades) defined in a service layer to satisfy business specific (and presentation-technology agnostic) use cases. In Spring, these service objects, any other business-specific objects, data access objects, etc. exist in a distinct 'business context', which contains no web or presentation layer objects (presentation objects such as Spring MVC controllers are typically configured in a distinct 'presentation context'). This section details how one configures a Spring container (a WebApplicationContext) that contains all of the 'business beans' in one’s application.

On to specifics: all that one need do is to declare a ContextLoaderListener in the standard Java EE servlet web.xml file of one’s web application, and add a contextConfigLocation<context-param/> section (in the same file) that defines which set of Spring XML configuration files to load.

Find below the <listener/> configuration:

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

Find below the <context-param/> configuration:

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/applicationContext\*.xml</param-value>

</context-param>

If you don’t specify the contextConfigLocation context parameter, the ContextLoaderListener will look for a file called /WEB-INF/applicationContext.xml to load. Once the context files are loaded, Spring creates a WebApplicationContext object based on the bean definitions and stores it in the ServletContext of the web application.

All Java web frameworks are built on top of the Servlet API, and so one can use the following code snippet to get access to this 'business context' ApplicationContext created by the ContextLoaderListener.

WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);

The WebApplicationContextUtils class is for convenience, so you don’t have to remember the name of the ServletContext attribute. Its getWebApplicationContext() method will return null if an object doesn’t exist under the WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE key. Rather than risk getting NullPointerExceptions in your application, it’s better to use the getRequiredWebApplicationContext() method. This method throws an exception when the ApplicationContext is missing.

Once you have a reference to the WebApplicationContext, you can retrieve beans by their name or type. Most developers retrieve beans by name and then cast them to one of their implemented interfaces.

Fortunately, most of the frameworks in this section have simpler ways of looking up beans. Not only do they make it easy to get beans from a Spring container, but they also allow you to use dependency injection on their controllers. Each web framework section has more detail on its specific integration strategies.

5.3. JSF

JavaServer Faces (JSF) is the JCP’s standard component-based, event-driven web user interface framework. As of Java EE 5, it is an official part of the Java EE umbrella.

For a popular JSF runtime as well as for popular JSF component libraries, check out the Apache MyFaces project. The MyFaces project also provides common JSF extensions such as MyFaces Orchestra: a Spring-based JSF extension that provides rich conversation scope support.

Spring Web Flow 2.0 provides rich JSF support through its newly established Spring Faces module, both for JSF-centric usage (as described in this section) and for Spring-centric usage (using JSF views within a Spring MVC dispatcher). Check out the Spring Web Flow website for details!

The key element in Spring’s JSF integration is the JSF ELResolver mechanism.

5.3.1. Spring Bean Resolver

SpringBeanFacesELResolver is a JSF 1.2+ compliant ELResolver implementation, integrating with the standard Unified EL as used by JSF 1.2 and JSP 2.1. Like SpringBeanVariableResolver, it delegates to the Spring’s 'business context' WebApplicationContext first, then to the default resolver of the underlying JSF implementation.

Configuration-wise, simply define SpringBeanFacesELResolver in your JSF faces-context.xml file:

<faces-config>

<application>

<el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>

...

</application>

</faces-config>

5.3.2. FacesContextUtils

A custom VariableResolver works well when mapping one’s properties to beans in faces-config.xml, but at times one may need to grab a bean explicitly. The FacesContextUtils class makes this easy. It is similar to WebApplicationContextUtils, except that it takes a FacesContext parameter rather than a ServletContext parameter.

ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());

5.4. Apache Struts 2.x

Invented by Craig McClanahan, Struts is an open source project hosted by the Apache Software Foundation. At the time, it greatly simplified the JSP/Servlet programming paradigm and won over many developers who were using proprietary frameworks. It simplified the programming model, it was open source (and thus free as in beer), and it had a large community, which allowed the project to grow and become popular among Java web developers.

Check out the Struts Spring Plugin for the built-in Spring integration shipped with Struts.

5.5. Tapestry 5.x

From the Tapestry homepage:

Tapestry is a "Component oriented framework for creating dynamic, robust, highly scalable web applications in Java."

While Spring has its own powerful web layer, there are a number of unique advantages to building an enterprise Java application using a combination of Tapestry for the web user interface and the Spring container for the lower layers.

For more information, check out Tapestry’s dedicated integration module for Spring.

5.6. Further Resources

Find below links to further resources about the various web frameworks described in this chapter.

The JSF homepage

The Struts homepage

The Tapestry homepage

## Web on Reactive Stack

Version 5.0.8.RELEASE

This part of the documentation covers support for reactive stack, web applications built on a Reactive Streams API to run on non-blocking servers such as Netty, Undertow, and Servlet 3.1+ containers. Individual chapters cover the Spring WebFlux framework, the reactive WebClient, support for Testing, and Reactive Libraries. For Servlet stack, web applications, please see Web on Servlet Stack.

1. Spring WebFlux

1.1. Introduction

The original web framework included in the Spring Framework, Spring Web MVC, was purpose built for the Servlet API and Servlet containers. The reactive stack, web framework, Spring WebFlux, was added later in version 5.0. It is fully non-blocking, supports Reactive Streams back pressure, and runs on servers such as Netty, Undertow, and Servlet 3.1+ containers.

Both web frameworks mirror the names of their source modules spring-webmvc and spring-webflux and co-exist side by side in the Spring Framework. Each module is optional. Applications may use one or the other module, or in some cases both — e.g. Spring MVC controllers with the reactive WebClient.

1.1.1. Motivation

Why was Spring WebFlux created?

Part of the answer is the need for a non-blocking web stack to handle concurrency with a small number of threads and scale with less hardware resources. Servlet 3.1 did provide an API for non-blocking I/O. However, using it leads away from the rest of the Servlet API where contracts are synchronous (Filter, Servlet) or blocking (getParameter, getPart). This was the motivation for a new common API to serve as a foundation across any non-blocking runtime. That is important because of servers such as Netty that are well established in the async, non-blocking space.

The other part of the answer is functional programming. Much like the addition of annotations in Java 5 created opportunities — e.g. annotated REST controllers or unit tests, the addition of lambda expressions in Java 8 created opportunities for functional APIs in Java. This is a boon for non-blocking applications and continuation style APIs — as popularized by CompletableFuture and ReactiveX, that allow declarative composition of asynchronous logic. At the programming model level Java 8 enabled Spring WebFlux to offer functional web endpoints alongside with annotated controllers.

1.1.2. Define "reactive"

We touched on non-blocking and functional but what does reactive mean?

The term "reactive" refers to programming models that are built around reacting to change — network component reacting to I/O events, UI controller reacting to mouse events, etc. In that sense non-blocking is reactive because instead of being blocked we are now in the mode of reacting to notifications as operations complete or data becomes available.

There is also another important mechanism that we on the Spring team associate with "reactive" and that is non-blocking back pressure. In synchronous, imperative code, blocking calls serve as a natural form of back pressure that forces the caller to wait. In non-blocking code it becomes important to control the rate of events so that a fast producer does not overwhelm its destination.

Reactive Streams is a small spec, also adopted in Java 9, that defines the interaction between asynchronous components with back pressure. For example a data repository — acting as Publisher, can produce data that an HTTP server — acting as Subscriber, can then write to the response. The main purpose of Reactive Streams is to allow the subscriber to control how fast or how slow the publisher will produce data.

Common question: what if a publisher can’t slow down?

The purpose of Reactive Streams is only to establish the mechanism and a boundary. If a publisher can’t slow down then it has to decide whether to buffer, drop, or fail.

1.1.3. Reactive API

Reactive Streams plays an important role for interoperability. It is of interest to libraries and infrastructure components but less useful as an application API because it is too low level. What applications need is a higher level and richer, functional API to compose async logic — similar to the Java 8 Stream API but not only for collections. This is the role that reactive libraries play.

Reactor is the reactive library of choice for Spring WebFlux. It provides the Mono and Flux API types to work on data sequences of 0..1 and 0..N through a rich set of operators aligned with the ReactiveX vocabulary of operators. Reactor is a Reactive Streams library and therefore all of its operators support non-blocking back pressure. Reactor has a strong focus on server-side Java. It is developed in close collaboration with Spring.

WebFlux requires Reactor as a core dependency but it is interoperable with other reactive libraries via Reactive Streams. As a general rule WebFlux APIs accept a plain Publisher as input, adapt it to Reactor types internally, use those, and then return either Flux or Mono as output. So you can pass any Publisher as input and you can apply operations on the output, but you’ll need to adapt the output for use with another reactive library. Whenever feasible — e.g. annotated controllers, WebFlux adapts transparently to the use of RxJava or other reactive library. See Reactive Libraries for more details.

1.1.4. Programming models

The spring-web module contains the reactive foundation that underlies Spring WebFlux including HTTP abstractions, Reactive Streams adapters for supported servers, codecs, and a core WebHandler API comparable to the Servlet API but with non-blocking contracts.

On that foundation Spring WebFlux provides a choice of two programming models:

Annotated Controllers — consistent with Spring MVC, and based on the same annotations from the spring-web module. Both Spring MVC and WebFlux controllers support reactive (Reactor, RxJava) return types and as a result it is not easy to tell them apart. One notable difference is that WebFlux also supports reactive @RequestBody arguments.

Functional Endpoints — lambda-based, lightweight, functional programming model. Think of this as a small library or a set of utilities that an application can use to route and handle requests. The big difference with annotated controllers is that the application is in charge of request handling from start to finish vs declaring intent through annotations and being called back.

1.1.5. Applicability

Spring MVC or WebFlux?

A natural question to ask but one that sets up an unsound dichotomy. It’s actually both working together to expand the range of available options. The two are designed for continuity and consistency with each other, they are available side by side, and feedback from each side benefits both sides. The diagram below shows how the two relate, what they have in common, and what each supports uniquely:

spring mvc and webflux venn

Below are some specific points to consider:

If you have a Spring MVC application that works fine, there is no need to change. Imperative programming is the easiest way to write, understand, and debug code. You have maximum choice of libraries since historically most are blocking.

If you are already shopping for a non-blocking web stack, Spring WebFlux offers the same execution model benefits as others in this space and also provides a choice of servers — Netty, Tomcat, Jetty, Undertow, Servlet 3.1+ containers, a choice of programming models — annotated controllers and functional web endpoints, and a choice of reactive libraries — Reactor, RxJava, or other.

If you are interested in a lightweight, functional web framework for use with Java 8 lambdas or Kotlin then use the Spring WebFlux functional web endpoints. That can also be a good choice for smaller applications or microservices with less complex requirements that can benefit from greater transparency and control.

In a microservice architecture you can have a mix of applications with either Spring MVC or Spring WebFlux controllers, or with Spring WebFlux functional endpoints. Having support for the same annotation-based programming model in both frameworks makes it easier to re-use knowledge while also selecting the right tool for the right job.

A simple way to evaluate an application is to check its dependencies. If you have blocking persistence APIs (JPA, JDBC), or networking APIs to use, then Spring MVC is the best choice for common architectures at least. It is technically feasible with both Reactor and RxJava to perform blocking calls on a separate thread but you wouldn’t be making the most of a non-blocking web stack.

If you have a Spring MVC application with calls to remote services, try the reactive WebClient. You can return reactive types (Reactor, RxJava, or other) directly from Spring MVC controller methods. The greater the latency per call, or the interdependency among calls, the more dramatic the benefits. Spring MVC controllers can call other reactive components too.

If you have a large team, keep in mind the steep learning curve in the shift to non-blocking, functional, and declarative programming. A practical way to start without a full switch is to use the reactive WebClient. Beyond that start small and measure the benefits. We expect that for a wide range of applications the shift is unnecessary. If you are unsure what benefits to look for, start by learning about how non-blocking I/O works (e.g. concurrency on single-threaded Node.js) and its effects.

1.1.6. Servers

Spring WebFlux is supported on Tomcat, Jetty, Servlet 3.1+ containers, as well as on non-Servlet runtimes such as Netty and Undertow. All servers are adapted to a low-level, common API so that higher level programming models can be supported across servers.

Spring WebFlux does not have built-in support to start or stop a server. However it is easy to assemble an application from Spring configuration, and WebFlux infrastructure, and run it with a few lines of code.

Spring Boot has a WebFlux starter that automates these steps. By default the starter uses Netty but it is easy to switch to Tomcat, Jetty, or Undertow simply by changing your Maven or Gradle dependencies. Spring Boot defaults to Netty because it is more widely used in the async, non-blocking space, and provides a client and a server share resources.

Tomcat and Jetty can be used with both Spring MVC and WebFlux. Keep in mind however that the way they’re used is very different. Spring MVC relies on Servlet blocking I/O and allows applications to use the Servlet API directly if they need to. Spring WebFlux relies on Servlet 3.1 non-blocking I/O and uses the Servlet API behind a low-level adapter and not exposed for direct use.

For Undertow, Spring WebFlux uses Undertow APIs directly without the Servlet API.

1.1.7. Performance vs scale

Performance has many characteristics and meanings. Reactive and non-blocking generally do not make applications run faster. They can, in some cases, for example if using the WebClient to execute remote calls in parallel. On the whole it requires more work to do things the non-blocking way and that can increase slightly the required processing time.

The key expected benefit of reactive and non-blocking is the ability to scale with a small, fixed number of threads and less memory. That makes applications more resilient under load because they scale in a more predictable way. In order to observe those benefits however you need to have some latency including a mix of slow and unpredictable network I/O. That’s where the reactive stack begins to show its strengths and the differences can be dramatic.

1.1.8. Concurrency Model

Both Spring MVC and Spring WebFlux support annotated controllers, but there is a key difference in the concurrency model and default assumptions for blocking and threads.

In Spring MVC, and servlet applications in general, it is assumed that applications may block the current thread, e.g. for remote calls, and for this reason servlet containers use a large thread pool, to absorb potential blocking during request handling.

In Spring WebFlux, and non-blocking servers in general, it is assumed that applications will not block, and therefore non-blocking servers use a small, fixed-size thread pool (event loop workers) to handle requests.

To "scale" and "small number of threads" may sound contradictory but to never block the current thread, and rely on callbacks instead, means you don’t need extra threads as there are no blocking calls to absorb.

Invoking a Blocking API

What if you do need to use a blocking library? Both Reactor and RxJava provide the publishOn operator to continue processing on a different thread. That means there is an easy escape latch. Keep in mind however that blocking APIs are not a good fit for this concurrency model.

Mutable State

In Reactor and RxJava, logic is declared through operators, and at runtime, a reactive pipeline is formed where data is processed sequentially, in distinct stages. A key benefit of that is that it frees applications from having to protect mutable state because application code within that pipeline is never invoked concurrently.

Threading Model

What threads should you expect to see on a server running with Spring WebFlux?

On a "vanilla" Spring WebFlux server (e.g. no data access, nor other optional dependencies), you can expect one thread for the server, and several others for request processing (typically as many as the number of CPU cores). Servlet containers, however, may start with more threads (e.g. 10 on Tomcat), in support of both servlet, blocking I/O and servlet 3.1, non-blocking I/O usage.

The reactive WebClient operates in event loop style. So you’ll see a small, fixed number of processing threads related to that, e.g. "reactor-http-nio-" with the Reactor Netty connector. However if Reactor Netty is used for both client and server, the two will share event loop resources by default.

Reactor and RxJava provide thread pool abstractions, called Schedulers, to use with the publishOn operator that is used to switch processing to a different thread pool. The schedulers have names that suggest a specific concurrency strategy, e.g. "parallel" for CPU-bound work with a limited number of threads, or "elastic" for I/O-bound work with a large number of threads. If you see such threads it means some code is using a specific thread pool Scheduler strategy.

Data access libraries and other 3rd party dependencies may also create and use threads of their own.

Configuring

The Spring Framework does not provide support for starting and stopping servers. To configure the threading model for a server, you’ll need to use server-specific config APIs, or if using Spring Boot, check the Spring Boot configuration options for each server. The WebClient can be configured directly. For all other libraries, refer to their respective documentation.

1.2. Reactive Core

The spring-web module contains abstractions and infrastructure to build reactive web applications. For server side processing this is organized in two distinct levels:

HttpHandler — basic, common API for HTTP request handling with non-blocking I/O and (Reactive Streams) back pressure, along with adapters for each supported server.

WebHandler API — slightly higher level, but still general purpose API for server request handling, which underlies higher level programming models such as annotated controllers and functional endpoints.

The reactive core also includes Codecs for client and server side use.

1.2.1. HttpHandler

HttpHandler is a simple contract with a single method to handle a request and response. It is intentionally minimal as its main purpose is to provide an abstraction over different server APIs for HTTP request handling.

Supported server APIs:

Server name Server API used Reactive Streams support

Netty

Netty API

Reactor Netty

Undertow

Undertow API

spring-web: Undertow to Reactive Streams bridge

Tomcat

Servlet 3.1 non-blocking I/O; Tomcat API to read and write ByteBuffers vs byte[]

spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge

Jetty

Servlet 3.1 non-blocking I/O; Jetty API to write ByteBuffers vs byte[]

spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge

Servlet 3.1 container

Servlet 3.1 non-blocking I/O

spring-web: Servlet 3.1 non-blocking I/O to Reactive Streams bridge

Server dependencies (and supported versions):

Server name Group id Artifact name

Reactor Netty

io.projectreactor.ipc

reactor-netty

Undertow

io.undertow

undertow-core

Tomcat

org.apache.tomcat.embed

tomcat-embed-core

Jetty

org.eclipse.jetty

jetty-server, jetty-servlet

Code snippets to adapt HttpHandler to each server API:

Reactor Netty

HttpHandler handler = ...

ReactorHttpHandlerAdapter adapter = new ReactorHttpHandlerAdapter(handler);

HttpServer.create(host, port).newHandler(adapter).block();

Undertow

HttpHandler handler = ...

UndertowHttpHandlerAdapter adapter = new UndertowHttpHandlerAdapter(handler);

Undertow server = Undertow.builder().addHttpListener(port, host).setHandler(adapter).build();

server.start();

Tomcat

HttpHandler handler = ...

Servlet servlet = new TomcatHttpHandlerAdapter(handler);

Tomcat server = new Tomcat();

File base = new File(System.getProperty("java.io.tmpdir"));

Context rootContext = server.addContext("", base.getAbsolutePath());

Tomcat.addServlet(rootContext, "main", servlet);

rootContext.addServletMappingDecoded("/", "main");

server.setHost(host);

server.setPort(port);

server.start();

Jetty

HttpHandler handler = ...

Servlet servlet = new JettyHttpHandlerAdapter(handler);

Server server = new Server();

ServletContextHandler contextHandler = new ServletContextHandler(server, "");

contextHandler.addServlet(new ServletHolder(servlet), "/");

contextHandler.start();

ServerConnector connector = new ServerConnector(server);

connector.setHost(host);

connector.setPort(port);

server.addConnector(connector);

server.start();

Servlet 3.1+ Container

To deploy as a WAR to any Servlet 3.1+ container, simply extend and include AbstractReactiveWebInitializer in the WAR, which wraps an HttpHandler with ServletHttpHandlerAdapter and registers that as a Servlet.

1.2.2. WebHandler API

The WebHandler API is a general purpose, server, web API for processing requests through a chain of WebExceptionHandler, WebFilter, and a target WebHandler components. The chain can be assembled with WebHttpHandlerBuilder either by adding components to the builder or by having them detected from a Spring ApplicationContext. The builder returns an HttpHandler that can then be used to run on any of the supported servers.

While HttpHandler aims to be the most minimal contract across HTTP servers, the WebHandler API provides essential features commonly used to build web applications. For example, the ServerWebExchange available to WebHandler API components provides access not only to the request and response, but also to request and session attributes, access to parsed form data, multipart data, and more.

Special bean types

The table below lists the components that WebHttpHandlerBuilder detects:

Bean name Bean type Count Description

<any>

WebExceptionHandler

0..N

Provide handling for exceptions from the chain of WebFilter's and the target WebHandler. For more details, see Exceptions.

<any>

WebFilter

0..N

Apply interception style logic to before and after the rest of the filter chain and the target WebHandler. For more details, see Filters.

"webHandler"

WebHandler

1

The handler for the request.

"webSessionManager"

WebSessionManager

0..1

The manager for WebSession's exposed through a method on ServerWebExchange. DefaultWebSessionManager by default.

"serverCodecConfigurer"

ServerCodecConfigurer

0..1

For access to HttpMessageReader's for parsing form data and multipart data that’s then exposed through methods on ServerWebExchange. ServerCodecConfigurer.create() by default.

"localeContextResolver"

LocaleContextResolver

0..1

The resolver for LocaleContext exposed through a method on ServerWebExchange. AcceptHeaderLocaleContextResolver by default.

Form data

ServerWebExchange exposes the following method for access to form data:

Mono<MultiValueMap<String, String>> getFormData();

The DefaultServerWebExchange uses the configured HttpMessageReader to parse form data ("application/x-www-form-urlencoded") into a MultiValueMap. By default FormHttpMessageReader is configured for use via the ServerCodecConfigurer bean (see Web Handler API).

Multipart data

Same in Spring MVC

ServerWebExchange exposes the following method for access to multipart data:

Mono<MultiValueMap<String, Part>> getMultipartData();

The DefaultServerWebExchange uses the configured HttpMessageReader<MultiValueMap<String, Part>> to parse "multipart/form-data" content into a MultiValueMap. At present Synchronoss NIO Multipart is the only 3rd party library supported, and the only library we know for non-blocking parsing of multipart requests. It is enabled through the ServerCodecConfigurer bean (see Web Handler API).

To parse multipart data in streaming fashion, use the Flux<Part> returned from an HttpMessageReader<Part> instead. For example in an annotated controller use of @RequestPart implies Map-like access to individual parts by name, and hence requires parsing multipart data in full. By contrast @RequestBody can be used to decode the content to Flux<Part> without collecting to a MultiValueMap.

1.2.3. Filters

Same in Spring MVC

In the WebHandler API, a WebFilter can be used to apply interception-style logic before and after the rest of the processing chain of filters and the target WebHandler. When using the WebFlux Config, registering a WebFilter is as simple as declaring it as a Spring bean, and optionally expressing precedence via @Order on the bean declaration or by implementing Ordered.

The following describe the available WebFilter implementations:

Forwarded Headers

Same in Spring MVC

As a request goes through proxies such as load balancers the host, port, and scheme may change presenting a challenge for applications that need to create links to resources since the links should reflect the host, port, and scheme of the original request as seen from a client perspective.

RFC 7239 defines the "Forwarded" HTTP header for proxies to use to provide information about the original request. There are also other non-standard headers in use such as "X-Forwarded-Host", "X-Forwarded-Port", and "X-Forwarded-Proto".

ForwardedHeaderFilter detects, extracts, and uses information from the "Forwarded" header, or from "X-Forwarded-Host", "X-Forwarded-Port", and "X-Forwarded-Proto". It wraps the request in order to overlay its host, port, and scheme and also "hides" the forwarded headers for subsequent processing.

Note that there are security considerations when using forwarded headers as explained in Section 8 of RFC 7239. At the application level it is difficult to determine whether forwarded headers can be trusted or not. This is why the network upstream should be configured correctly to filter out untrusted forwarded headers from the outside.

Applications that don’t have a proxy and don’t need to use forwarded headers can configure the ForwardedHeaderFilter to remove and ignore such headers.

CORS

Same in Spring MVC

Spring WebFlux provides fine-grained support for CORS configuration through annotations on controllers. However when used with Spring Security it is advisable to rely on the built-in CorsFilter that must be ordered ahead of Spring Security’s chain of filters.

See the section on CORS and the CORS WebFilter for more details.

1.2.4. Exceptions

Same in Spring MVC

In the WebHandler API, a WebExceptionHandler can be used to to handle exceptions from the chain of WebFilter's and the target WebHandler. When using the WebFlux Config, registering a WebExceptionHandler is as simple as declaring it as a Spring bean, and optionally expressing precedence via @Order on the bean declaration or by implementing Ordered.

Below are the available WebExceptionHandler implementations:

Exception Handler Description

ResponseStatusExceptionHandler

Provides handling for exceptions of type ResponseStatusException by setting the response to the HTTP status code of the exception.

WebFluxResponseStatusExceptionHandler

Extension of ResponseStatusExceptionHandler that can also determine the HTTP status code an @ResponseStatus annotation on any exception.

This handler is declared in the WebFlux Config.

1.2.5. Codecs

Same in Spring MVC

HttpMessageReader and HttpMessageWriter are contracts for encoding and decoding HTTP request and response content via non-blocking I/O with (Rective Streams) back pressure.

Encoder and Decoder are contracts for encoding and decoding content, independent of HTTP. They can be wrapped with EncoderHttpMessageWriter or DecoderHttpMessageReader and used for web processing.

All codecs are for client or server side use. All build on DataBuffer which abstracts byte buffer representations such as the Netty ByteBuf or java.nio.ByteBuffer (see Data Buffers and Codecs for more details). ClientCodecConfigurer and ServerCodecConfigurer are typically used to configure and customize the codecs to use in an application.

The spring-core module has encoders and decoders for byte[], ByteBuffer, DataBuffer, Resource, and String. The spring-web module adds encoders and decoders for Jackson JSON, Jackson Smile, JAXB2, along with other web-specific HTTP message readers and writers for form data, multipart requests, and server-sent events.

Jackson

The decoder relies on Jackson’s non-blocking, byte array parser to parse a stream of byte chunks into a TokenBuffer stream, which can then be turned into Objects with Jackson’s ObjectMapper. JSON and Smile (binary JSON) data formats are currently supported.

The encoder processes a Publisher<?> as follows:

if the Publisher is a Mono (i.e. single value), the value is encoded when available.

if media type is application/stream+json for JSON or application/stream+x-jackson-smile for Smile, each value produced by the Publisher is encoded individually (and followed by a new line in JSON).

otherwise all items from the Publisher are gathered in with Flux#collectToList() and the resulting collection is encoded as an array.

As a special case to the above rules the ServerSentEventHttpMessageWriter feeds items emitted from its input Publisher individually into the Jackson2JsonEncoder as a Mono<?>.

Note that both the Jackson JSON encoder and decoder explicitly back out of rendering elements of type String. Instead String's are treated as low level content, (i.e. serialized JSON) and are rendered as-is by the CharSequenceEncoder. If you want a Flux<String> rendered as a JSON array, you’ll have to use Flux#collectToList() and provide a Mono<List<String>> instead.

HTTP Streaming

Same in Spring MVC

When a multi-value, reactive type such as Flux is used for response rendering, it may be collected to a List and rendered as a whole (e.g. JSON array), or it may be treated as an infinite stream with each item flushed immediately. The determination for which is which is made based on content negotiation and the selected media type which may imply a streaming format (e.g. "text/event-stream", "application/stream+json"), or not (e.g. "application/json").

When streaming to the HTTP response, regardless of the media type (e.g. text/event-stream, application/stream+json), it is important to send data periodically, since the write would fail if the client has disconnected. The send could take the form of an empty (comment-only) SSE event, or any other data that the other side would have to interpret as a heartbeat and ignore.

1.3. DispatcherHandler

Same in Spring MVC

Spring WebFlux, like Spring MVC, is designed around the front controller pattern where a central WebHandler, the DispatcherHandler, provides a shared algorithm for request processing while actual work is performed by configurable, delegate components. This model is flexible and supports diverse workflows.

DispatcherHandler discovers the delegate components it needs from Spring configuration. It is also designed to be a Spring bean itself and implements ApplicationContextAware for access to the context it runs in. If DispatcherHandler is declared with the bean name "webHandler" it is in turn discovered by WebHttpHandlerBuilder which puts together a request processing chain as described in WebHandler API.

Spring configuration in a WebFlux application typically contains:

DispatcherHandler with the bean name "webHandler"

WebFilter and WebExceptionHandler beans

DispatcherHandler special beans

Others

The configuration is given to WebHttpHandlerBuilder to build the processing chain:

ApplicationContext context = ...

HttpHandler handler = WebHttpHandlerBuilder.applicationContext(context);

The resulting HttpHandler is ready for use with a server adapter.

1.3.1. Special bean types

Same in Spring MVC

The DispatcherHandler delegates to special beans to process requests and render the appropriate responses. By "special beans" we mean Spring-managed, Object instances that implement WebFlux framework contracts. Those usually come with built-in contracts but you can customize their properties, extend then, or replaced.

The table below lists the special beans detected by the DispatcherHandler. Note that there are also some other beans detected at a lower level, see Special bean types in the Web Handler API.

Bean type Explanation

HandlerMapping

Map a request to a handler. The mapping is based on some criteria the details of which vary by HandlerMapping implementation — annotated controllers, simple URL pattern mappings, etc.

The main HandlerMapping implementations are RequestMappingHandlerMapping for @RequestMapping annotated methods, RouterFunctionMapping for functional endpoint routes, and SimpleUrlHandlerMapping for explicit registrations of URI path patterns and WebHandler's.

HandlerAdapter

Help the DispatcherHandler to invoke a handler mapped to a request regardless of how the handler is actually invoked. For example invoking an annotated controller requires resolving annotations. The main purpose of a HandlerAdapter is to shield the DispatcherHandler from such details.

HandlerResultHandler

Process the result from the handler invocation and finalize the response. See Result Handling.

1.3.2. WebFlux Config

Same in Spring MVC

Applications can declare the infrastructure beans listed under Web Handler API and DispatcherHandler that are required to process requests. However in most cases the WebFlux Config is the best starting point. It declares the required beans and provides a higher level configuration callback API to customize it.

Spring Boot relies on the WebFlux config to configure Spring WebFlux and also provides many extra convenient options.

1.3.3. Processing

Same in Spring MVC

The DispatcherHandler processes requests as follows:

Each HandlerMapping is asked to find a matching handler and the first match is used.

If a handler is found, it is executed through an appropriate HandlerAdapter which exposes the return value from the execution as HandlerResult.

The HandlerResult is given to an appropriate HandlerResultHandler to complete processing by writing to the response directly or using a view to render.

1.3.4. Result Handling

The return value from the invocation of a handler, through a HandlerAdapter, is wrapped as HandlerResult, along with some additional context, and passed to the first HandlerResultHandler that claims support for it. The table below shows the available HandlerResultHandler implementations all of which are declared in the WebFlux Config:

Result Handler Type Return Values Default Order

ResponseEntityResultHandler

ResponseEntity, typically from @Controller's.

0

ServerResponseResultHandler

ServerResponse, typically from functional endpoints.

0

ResponseBodyResultHandler

Handle return values from @ResponseBody methods or @RestController classes.

100

ViewResolutionResultHandler

CharSequence or View, Model or Map, Rendering, or any other Object is treated as a model attribute.

Also see View Resolution.

Integer.MAX\_VALUE

1.3.5. Exceptions

Same in Spring MVC

The HandlerResult returned from a HandlerAdapter may expose a function for error handling based on some handler-specific mechanism. This error function is called if:

the handler (e.g. @Controller) invocation fails.

handling of the handler return value through a HandlerResultHandler fails.

The error function can change the response, e.g. to an error status, as long as an error signal occurs before the reactive type returned from the handler produces any data items.

This is how @ExceptionHandler methods in @Controller classes are supported. By contrast, support for the same in Spring MVC is built on a HandlerExceptionResolver. This generally shouldn’t matter, however, keep in mind that in WebFlux you cannot use a @ControllerAdvice to handle exceptions that occur before a handler is chosen.

See also Exceptions in the Annotated Controller section, or Exceptions in the WebHandler API section.

1.3.6. View Resolution

Same in Spring MVC

View resolution enables rendering to a browser with an HTML template and a model without tying you to a specific view technology. In Spring WebFlux, view resolution is supported through a dedicated HandlerResultHandler that uses ViewResolver's to map a String, representing a logical view name, to a View instance. The View is then used to render the response.

Handling

Same in Spring MVC

The HandlerResult passed into ViewResolutionResultHandler contains the return value from the handler, and also the model that contains attributes added during request handling. The return value is processed as one of the following:

String, CharSequence — a logical view name to be resolved to a View through the list of configured ViewResolver's.

void — select a default view name based on the request path minus the leading and trailing slash, and resolve it to a View. The same also happens when a view name was not provided, e.g. model attribute was returned, or an async return value, e.g. Mono completed empty.

Rendering — API for view resolution scenarios; explore the options in your IDE with code completion.

Model, Map — extra model attributes to be added to the model for the request.

Any other — any other return value (except for simple types, as determined by BeanUtils#isSimpleProperty) is treated as a model attribute to be added to the model. The attribute name is derived from the Class name, using Conventions, unless a handler method @ModelAttribute annotation is present.

The model can contain asynchronous, reactive types (e.g. from Reactor, RxJava). Prior to rendering, AbstractView resolves such model attributes into concrete values and updates the model. Single-value reactive types are resolved to a single value, or no value (if empty) while multi-value reactive types, e.g. Flux<T> are collected and resolved to List<T>.

To configure view resolution is as simple as adding a ViewResolutionResultHandler bean to your Spring configuration. WebFlux Config provides a dedicated configuration API for view resolution.

See View Technologies for more on the view technologies integrated with Spring WebFlux.

Redirecting

Same in Spring MVC

The special redirect: prefix in a view name allows you to perform a redirect. The UrlBasedViewResolver (and sub-classes) recognize this as an instruction that a redirect is needed. The rest of the view name is the redirect URL.

The net effect is the same as if the controller had returned a RedirectView or Rendering.redirectTo("abc").build(), but now the controller itself can simply operate in terms of logical view names. A view name such as redirect:/some/resource is relative to the current application, while the view name redirect:http://example.com/arbitrary/path redirects to an absolute URL.

Content negotiation

Same in Spring MVC

ViewResolutionResultHandler supports content negotiation. It compares the request media type(s) with the media type(s) supported by each selected View. The first View that supports the requested media type(s) is used.

In order to support media types such as JSON and XML, Spring WebFlux provides HttpMessageWriterView which is a special View that renders through an HttpMessageWriter. Typically you would configure these as default views through the WebFlux Config. Default views are always selected and used if they match the requested media type.

1.4. Annotated Controllers

Same in Spring MVC

Spring WebFlux provides an annotation-based programming model where @Controller and @RestController components use annotations to express request mappings, request input, exception handling, and more. Annotated controllers have flexible method signatures and do not have to extend base classes nor implement specific interfaces.

Here is a basic example:

@RestController

public class HelloController {

@GetMapping("/hello")

public String handle() {

return "Hello WebFlux";

}

}

In this example the methods returns a String to be written to the response body.

1.4.1. @Controller

Same in Spring MVC

You can define controller beans using a standard Spring bean definition. The @Controller stereotype allows for auto-detection, aligned with Spring general support for detecting @Component classes in the classpath and auto-registering bean definitions for them. It also acts as a stereotype for the annotated class, indicating its role as a web component.

To enable auto-detection of such @Controller beans, you can add component scanning to your Java configuration:

@Configuration

@ComponentScan("org.example.web")

public class WebConfig {

// ...

}

@RestController is a composed annotation that is itself meta-annotated with @Controller and @ResponseBody indicating a controller whose every method inherits the type-level @ResponseBody annotation and therefore writes directly to the response body vs view resolution and rendering with an HTML template.

1.4.2. Request Mapping

Same in Spring MVC

The @RequestMapping annotation is used to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types. It can be used at the class-level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.

There are also HTTP method specific shortcut variants of @RequestMapping:

@GetMapping

@PostMapping

@PutMapping

@DeleteMapping

@PatchMapping

The above are Custom Annotations that are provided out of the box because arguably most controller methods should be mapped to a specific HTTP method vs using @RequestMapping which by default matches to all HTTP methods. At the same an @RequestMapping is still needed at the class level to express shared mappings.

Below is an example with type and method level mappings:

@RestController

@RequestMapping("/persons")

class PersonController {

@GetMapping("/{id}")

public Person getPerson(@PathVariable Long id) {

// ...

}

@PostMapping

@ResponseStatus(HttpStatus.CREATED)

public void add(@RequestBody Person person) {

// ...

}

}

URI Patterns

Same in Spring MVC

You can map requests using glob patterns and wildcards:

? matches one character

\* matches zero or more characters within a path segment

\*\* match zero or more path segments

You can also declare URI variables and access their values with @PathVariable:

@GetMapping("/owners/{ownerId}/pets/{petId}")

public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {

// ...

}

URI variables can be declared at the class and method level:

@Controller

@RequestMapping("/owners/{ownerId}")

public class OwnerController {

@GetMapping("/pets/{petId}")

public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {

// ...

}

}

URI variables are automatically converted to the appropriate type or`TypeMismatchException` is raised. Simple types — int, long, Date, are supported by default and you can register support for any other data type. See Type Conversion and DataBinder.

URI variables can be named explicitly — e.g. @PathVariable("customId"), but you can leave that detail out if the names are the same and your code is compiled with debugging information or with the -parameters compiler flag on Java 8.

The syntax {\*varName} declares a URI variable that matches zero or more remaining path segments. For example /resources/{\*path} matches all files /resources/ and the "path" variable captures the complete relative path.

The syntax {varName:regex} declares a URI variable with a regular expressions with the syntax {varName:regex} — e.g. given URL "/spring-web-3.0.5 .jar", the below method extracts the name, version, and file extension:

@GetMapping("/{name:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{ext:\\.[a-z]+}")

public void handle(@PathVariable String version, @PathVariable String ext) {

// ...

}

URI path patterns can also have embedded ${…​} placeholders that are resolved on startup via PropertyPlaceHolderConfigurer against local, system, environment, and other property sources. This can be used for example to parameterize a base URL based on some external configuration.

Spring WebFlux uses PathPattern and the PathPatternParser for URI path matching support both of which are located in spring-web and expressly designed for use with HTTP URL paths in web applications where a large number of URI path patterns are matched at runtime.

Spring WebFlux does not support suffix pattern matching — unlike Spring MVC, where a mapping such as /person also matches to /person.\*. For URL based content negotiation, if needed, we recommend using a query parameter, which is simpler, more explicit, and less vulnerable to URL path based exploits.

Pattern Comparison

Same in Spring MVC

When multiple patterns match a URL, they must be compared to find the best match. This is done with PathPattern.SPECIFICITY\_COMPARATOR which looks for patterns that more specific.

For every pattern, a score is computed based the number of URI variables and wildcards where a URI variable scores lower than a wildcard. A pattern with a lower total score wins. If two patterns have the same score, then the longer is chosen.

Catch-all patterns, e.g. \*\*, {\*varName}, are excluded from the scoring and are always sorted last instead. If two patterns are both catch-all, the longer is chosen.

Consumable Media Types

Same in Spring MVC

You can narrow the request mapping based on the Content-Type of the request:

@PostMapping(path = "/pets", consumes = "application/json")

public void addPet(@RequestBody Pet pet) {

// ...

}

The consumes attribute also supports negation expressions — e.g. !text/plain means any content type other than "text/plain".

You can declare a shared consumes attribute at the class level. Unlike most other request mapping attributes however when used at the class level, a method-level consumes attribute overrides rather than extend the class level declaration.

MediaType provides constants for commonly used media types — e.g. APPLICATION\_JSON\_VALUE, APPLICATION\_XML\_VALUE.

Producible Media Types

Same in Spring MVC

You can narrow the request mapping based on the Accept request header and the list of content types that a controller method produces:

@GetMapping(path = "/pets/{petId}", produces = "application/json;charset=UTF-8")

@ResponseBody

public Pet getPet(@PathVariable String petId) {

// ...

}

The media type can specify a character set. Negated expressions are supported — e.g. !text/plain means any content type other than "text/plain".

For JSON content type, the UTF-8 charset should be specified even if RFC7159 clearly states that "no charset parameter is defined for this registration" because some browsers require it for interpreting correctly UTF-8 special characters.

You can declare a shared produces attribute at the class level. Unlike most other request mapping attributes however when used at the class level, a method-level produces attribute overrides rather than extend the class level declaration.

MediaType provides constants for commonly used media types — e.g. APPLICATION\_JSON\_UTF8\_VALUE, APPLICATION\_XML\_VALUE.

Parameters and Headers

Same in Spring MVC

You can narrow request mappings based on query parameter conditions. You can test for the presence of a query parameter ("myParam"), for the absence ("!myParam"), or for a specific value ("myParam=myValue"):

@GetMapping(path = "/pets/{petId}", params = "myParam=myValue")

public void findPet(@PathVariable String petId) {

// ...

}

You can also use the same with request header conditions:

@GetMapping(path = "/pets", headers = "myHeader=myValue")

public void findPet(@PathVariable String petId) {

// ...

}

HTTP HEAD, OPTIONS

Same in Spring MVC

@GetMapping — and also @RequestMapping(method=HttpMethod.GET), support HTTP HEAD transparently for request mapping purposes. Controller methods don’t need to change. A response wrapper, applied in the HttpHandler server adapter, ensures a "Content-Length" header is set to the number of bytes written and without actually writing to the response.

By default HTTP OPTIONS is handled by setting the "Allow" response header to the list of HTTP methods listed in all @RequestMapping methods with matching URL patterns.

For a @RequestMapping without HTTP method declarations, the "Allow" header is set to "GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS". Controller methods should always declare the supported HTTP methods for example by using the HTTP method specific variants — @GetMapping, @PostMapping, etc.

@RequestMapping method can be explicitly mapped to HTTP HEAD and HTTP OPTIONS, but that is not necessary in the common case.

Custom Annotations

Same in Spring MVC

Spring WebFlux supports the use of composed annotations for request mapping. Those are annotations that are themselves meta-annotated with @RequestMapping and composed to redeclare a subset (or all) of the @RequestMapping attributes with a narrower, more specific purpose.

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, and @PatchMapping are examples of composed annotations. They’re provided out of the box because arguably most controller methods should be mapped to a specific HTTP method vs using @RequestMapping which by default matches to all HTTP methods. If you need an example of composed annotations, look at how those are declared.

Spring WebFlux also supports custom request mapping attributes with custom request matching logic. This is a more advanced option that requires sub-classing RequestMappingHandlerMapping and overriding the getCustomMethodCondition method where you can check the custom attribute and return your own RequestCondition.

1.4.3. Handler methods

Same in Spring MVC

@RequestMapping handler methods have a flexible signature and can choose from a range of supported controller method arguments and return values.

Method arguments

Same in Spring MVC

The table below shows supported controller method arguments.

Reactive types (Reactor, RxJava, or other) are supported on arguments that require blocking I/O, e.g. reading the request body, to be resolved. This is marked in the description column. Reactive types are not expected on arguments that don’t require blocking.

JDK 1.8’s java.util.Optional is supported as a method argument in combination with annotations that have a required attribute — e.g. @RequestParam, @RequestHeader, etc, and is equivalent to required=false.

Controller method argument Description

ServerWebExchange

Access to the full ServerWebExchange — container for the HTTP request and response, request and session attributes, checkNotModified methods, and others.

ServerHttpRequest, ServerHttpResponse

Access to the HTTP request or response.

WebSession

Access to the session; this does not force the start of a new session unless attributes are added. Supports reactive types.

java.security.Principal

Currently authenticated user; possibly a specific Principal implementation class if known. Supports reactive types.

org.springframework.http.HttpMethod

The HTTP method of the request.

java.util.Locale

The current request locale, determined by the most specific LocaleResolver available, in effect, the configured LocaleResolver/LocaleContextResolver.

java.util.TimeZone + java.time.ZoneId

The time zone associated with the current request, as determined by a LocaleContextResolver.

@PathVariable

For access to URI template variables. See URI Patterns.

@MatrixVariable

For access to name-value pairs in URI path segments. See Matrix variables.

@RequestParam

For access to Servlet request parameters. Parameter values are converted to the declared method argument type. See @RequestParam.

Note that use of @RequestParam is optional, e.g. to set its attributes. See "Any other argument" further below in this table.

@RequestHeader

For access to request headers. Header values are converted to the declared method argument type. See @RequestHeader.

@CookieValue

For access to cookies. Cookies values are converted to the declared method argument type. See @CookieValue.

@RequestBody

For access to the HTTP request body. Body content is converted to the declared method argument type using HttpMessageReader's. Supports reactive types. See @RequestBody.

HttpEntity<B>

For access to request headers and body. The body is converted with HttpMessageReader's. Supports reactive types. See HttpEntity.

@RequestPart

For access to a part in a "multipart/form-data" request. Supports reactive types. See Multipart and Multipart data.

java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap

For access to the model that is used in HTML controllers and exposed to templates as part of view rendering.

@ModelAttribute

For access to an existing attribute in the model (instantiated if not present) with data binding and validation applied. See @ModelAttribute as well as Model and DataBinder.

Note that use of @ModelAttribute is optional, e.g. to set its attributes. See "Any other argument" further below in this table.

Errors, BindingResult

For access to errors from validation and data binding for a command object (i.e. @ModelAttribute argument), or errors from the validation of an @RequestBody or @RequestPart arguments; an Errors, or BindingResult argument must be declared immediately after the validated method argument.

SessionStatus + class-level @SessionAttributes

For marking form processing complete which triggers cleanup of session attributes declared through a class-level @SessionAttributes annotation. See @SessionAttributes for more details.

UriComponentsBuilder

For preparing a URL relative to the current request’s host, port, scheme, context path, and the literal part of the servlet mapping also taking into account Forwarded and X-Forwarded-\* headers. // TODO: See URI Links.

@SessionAttribute

For access to any session attribute; in contrast to model attributes stored in the session as a result of a class-level @SessionAttributes declaration. See @SessionAttribute for more details.

@RequestAttribute

For access to request attributes. See @RequestAttribute for more details.

Any other argument

If a method argument is not matched to any of the above, by default it is resolved as an @RequestParam if it is a simple type, as determined by BeanUtils#isSimpleProperty, or as an @ModelAttribute otherwise.

Return values

Same in Spring MVC

The table below shows supported controller method return values. Note that reactive types from libraries such as Reactor, RxJava, or other are generally supported for all return values.

Controller method return value Description

@ResponseBody

The return value is encoded through HttpMessageWriter's and written to the response. See @ResponseBody.

HttpEntity<B>, ResponseEntity<B>

The return value specifies the full response including HTTP headers and body be encoded through HttpMessageWriter's and written to the response. See ResponseEntity.

HttpHeaders

For returning a response with headers and no body.

String

A view name to be resolved with ViewResolver's and used together with the implicit model — determined through command objects and @ModelAttribute methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).

View

A View instance to use for rendering together with the implicit model — determined through command objects and @ModelAttribute methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).

java.util.Map, org.springframework.ui.Model

Attributes to be added to the implicit model with the view name implicitly determined based on the request path.

@ModelAttribute

An attribute to be added to the model with the view name implicitly determined based on the request path.

Note that @ModelAttribute is optional. See "Any other return value" further below in this table.

Rendering

An API for model and view rendering scenarios.

void

A method with a void, possibly async (e.g. Mono<Void>), return type (or a null return value) is considered to have fully handled the response if it also has a ServerHttpResponse, or a ServerWebExchange argument, or an @ResponseStatus annotation. The same is true also if the controller has made a positive ETag or lastModified timestamp check. // TODO: See Controllers for details.

If none of the above is true, a void return type may also indicate "no response body" for REST controllers, or default view name selection for HTML controllers.

Flux<ServerSentEvent>, Observable<ServerSentEvent>, or other reactive type

Emit server-sent events; the SeverSentEvent wrapper can be omitted when only data needs to be written (however text/event-stream must be requested or declared in the mapping through the produces attribute).

Any other return value

If a return value is not matched to any of the above, by default it is treated as a view name, if it is String or void (default view name selection applies); or as a model attribute to be added to the model, unless it is a simple type, as determined by BeanUtils#isSimpleProperty in which case it remains unresolved.

Type Conversion

Same in Spring MVC

Some annotated controller method arguments that represent String-based request input — e.g. @RequestParam, @RequestHeader, @PathVariable, @MatrixVariable, and @CookieValue, may require type conversion if the argument is declared as something other than String.

For such cases type conversion is automatically applied based on the configured converters. By default simple types such as int, long, Date, etc. are supported. Type conversion can be customized through a WebDataBinder, see [mvc-ann-initbinder], or by registering Formatters with the FormattingConversionService, see Spring Field Formatting.

Matrix variables

Same in Spring MVC

RFC 3986 discusses name-value pairs in path segments. In Spring WebFlux we refer to those as "matrix variables" based on an "old post" by Tim Berners-Lee but they can be also be referred to as URI path parameters.

Matrix variables can appear in any path segment, each variable separated by semicolon and multiple values separated by comma, e.g. "/cars;color=red,green;year=2012". Multiple values can also be specified through repeated variable names, e.g. "color=red;color=green;color=blue".

Unlike Spring MVC, in WebFlux the presence or absence of matrix variables in a URL does not affect request mappings. In other words you’re not required to use a URI variable to mask variable content. That said if you want to access matrix variables from a controller method you need to add a URI variable to the path segment where matrix variables are expected. Below is an example:

// GET /pets/42;q=11;r=22

@GetMapping("/pets/{petId}")

public void findPet(@PathVariable String petId, @MatrixVariable int q) {

// petId == 42

// q == 11

}

Given that all path segments may contain matrix variables, sometimes you may need to disambiguate which path variable the matrix variable is expected to be in. For example:

// GET /owners/42;q=11/pets/21;q=22

@GetMapping("/owners/{ownerId}/pets/{petId}")

public void findPet(

@MatrixVariable(name="q", pathVar="ownerId") int q1,

@MatrixVariable(name="q", pathVar="petId") int q2) {

// q1 == 11

// q2 == 22

}

A matrix variable may be defined as optional and a default value specified:

// GET /pets/42

@GetMapping("/pets/{petId}")

public void findPet(@MatrixVariable(required=false, defaultValue="1") int q) {

// q == 1

}

To get all matrix variables, use a MultiValueMap:

// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@GetMapping("/owners/{ownerId}/pets/{petId}")

public void findPet(

@MatrixVariable MultiValueMap<String, String> matrixVars,

@MatrixVariable(pathVar="petId"") MultiValueMap<String, String> petMatrixVars) {

// matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]

// petMatrixVars: ["q" : 22, "s" : 23]

}

@RequestParam

Same in Spring MVC

Use the @RequestParam annotation to bind query parameters to a method argument in a controller. The following code snippet shows the usage:

@Controller

@RequestMapping("/pets")

public class EditPetForm {

// ...

@GetMapping

public String setupForm(@RequestParam("petId") int petId, Model model) {

Pet pet = this.clinic.loadPet(petId);

model.addAttribute("pet", pet);

return "petForm";

}

// ...

}

Unlike the Servlet API "request paramater" concept that conflate query parameters, form data, and multiparts into one, in WebFlux each is accessed individually through the ServerWebExchange. While @RequestParam binds to query parameters only, you can use data binding to apply query paramerters, form data, and multiparts to a command object.

Method parameters using using the @RequestParam annotation are required by default, but you can specify that a method parameter is optional by setting @RequestParam's required flag to false or by declaring the argument with an java.util.Optional wrapper.

Type conversion is applied automatically if the target method parameter type is not String. See [mvc-ann-typeconversion].

When an @RequestParam annotation is declared as Map<String, String> or MultiValueMap<String, String> argument, the map is populated with all query parameters.

Note that use of @RequestParam is optional, e.g. to set its attributes. By default any argument that is a simple value type, as determined by BeanUtils#isSimpleProperty, and is not resolved by any other argument resolver, is treated as if it was annotated with @RequestParam.

@RequestHeader

Same in Spring MVC

Use the @RequestHeader annotation to bind a request header to a method argument in a controller.

Given request with headers:

Host localhost:8080

Accept text/html,application/xhtml+xml,application/xml;q=0.9

Accept-Language fr,en-gb;q=0.7,en;q=0.3

Accept-Encoding gzip,deflate

Accept-Charset ISO-8859-1,utf-8;q=0.7,\*;q=0.7

Keep-Alive 300

The following gets the value of the Accept-Encoding and Keep-Alive headers:

@GetMapping("/demo")

public void handle(

@RequestHeader("Accept-Encoding") String encoding,

@RequestHeader("Keep-Alive") long keepAlive) {

//...

}

Type conversion is applied automatically if the target method parameter type is not String. See [mvc-ann-typeconversion].

When an @RequestHeader annotation is used on a Map<String, String>, MultiValueMap<String, String>, or HttpHeaders argument, the map is populated with all header values.

Built-in support is available for converting a comma-separated string into an array/collection of strings or other types known to the type conversion system. For example a method parameter annotated with @RequestHeader("Accept") may be of type String but also String[] or List<String>.

@CookieValue

Same in Spring MVC

Use the @CookieValue annotation to bind the value of an HTTP cookie to a method argument in a controller.

Given request with the following cookie:

JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84

The following code sample demonstrates how to get the cookie value:

@GetMapping("/demo")

public void handle(@CookieValue("JSESSIONID") String cookie) {

//...

}

Type conversion is applied automatically if the target method parameter type is not String. See [mvc-ann-typeconversion].

@ModelAttribute

Same in Spring MVC

Use the @ModelAttribute annotation on a method argument to access an attribute from the model, or have it instantiated if not present. The model attribute is also overlaid with values of query parameters and form fields whose names match to field names. This is referred to as data binding and it saves you from having to deal with parsing and converting individual query parameters and form fields. For example:

@PostMapping("/owners/{ownerId}/pets/{petId}/edit")

public String processSubmit(@ModelAttribute Pet pet) { }

The Pet instance above is resolved as follows:

From the model if already added via Model.

From the HTTP session via @SessionAttributes.

From the invocation of a default constructor.

From the invocation of a "primary constructor" with arguments matching to query parameters or form fields; argument names are determined via JavaBeans @ConstructorProperties or via runtime-retained parameter names in the bytecode.

After the model attribute instance is obtained, data binding is applied. The WebExchangeDataBinder class matches names of query parameters and form fields to field names on the target Object. Matching fields are populated after type conversion is applied where necessary. For more on data binding (and validation) see Validation. For more on customizing data binding see DataBinder.

Data binding may result in errors. By default a WebExchangeBindException is raised but to check for such errors in the controller method, add a BindingResult argument immediately next to the @ModelAttribute as shown below:

@PostMapping("/owners/{ownerId}/pets/{petId}/edit")

public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {

if (result.hasErrors()) {

return "petForm";

}

// ...

}

Validation can be applied automatically after data binding by adding the javax.validation.Valid annotation or Spring’s @Validated annotation (also see Bean validation and Spring validation). For example:

@PostMapping("/owners/{ownerId}/pets/{petId}/edit")

public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult result) {

if (result.hasErrors()) {

return "petForm";

}

// ...

}

Spring WebFlux, unlike Spring MVC, supports reactive types in the model, e.g. Mono<Account> or io.reactivex.Single<Account>. An @ModelAttribute argument can be declared with or without a reactive type wrapper, and it will be resolved accordingly, to the actual value if necessary. Note however that in order to use a BindingResult argument, you must declare the @ModelAttribute argument before it without a reactive type wrapper, as shown earlier. Alternatively, you can handle any errors through the reactive type:

@PostMapping("/owners/{ownerId}/pets/{petId}/edit")

public Mono<String> processSubmit(@Valid @ModelAttribute("pet") Mono<Pet> petMono) {

return petMono

.flatMap(pet -> {

// ...

})

.onErrorResume(ex -> {

// ...

});

}

Note that use of @ModelAttribute is optional, e.g. to set its attributes. By default any argument that is not a simple value type, as determined by BeanUtils#isSimpleProperty, and is not resolved by any other argument resolver, is treated as if it was annotated with @ModelAttribute.

@SessionAttributes

Same in Spring MVC

@SessionAttributes is used to store model attributes in the WebSession between requests. It is a type-level annotation that declares session attributes used by a specific controller. This will typically list the names of model attributes or types of model attributes which should be transparently stored in the session for subsequent requests to access.

For example:

@Controller

@SessionAttributes("pet")

public class EditPetForm {

// ...

}

On the first request when a model attribute with the name "pet" is added to the model, it is automatically promoted to and saved in the WebSession. It remains there until another controller method uses a SessionStatus method argument to clear the storage:

@Controller

@SessionAttributes("pet")

public class EditPetForm {

// ...

@PostMapping("/pets/{id}")

public String handle(Pet pet, BindingResult errors, SessionStatus status) {

if (errors.hasErrors) {

// ...

}

status.setComplete();

// ...

}

}

}

@SessionAttribute

Same in Spring MVC

If you need access to pre-existing session attributes that are managed globally, i.e. outside the controller (e.g. by a filter), and may or may not be present use the @SessionAttribute annotation on a method parameter:

@GetMapping("/")

public String handle(@SessionAttribute User user) {

// ...

}

For use cases that require adding or removing session attributes consider injecting WebSession into the controller method.

For temporary storage of model attributes in the session as part of a controller workflow consider using SessionAttributes as described in @SessionAttributes.

@RequestAttribute

Same in Spring MVC

Similar to @SessionAttribute the @RequestAttribute annotation can be used to access pre-existing request attributes created earlier, e.g. by a WebFilter:

@GetMapping("/")

public String handle(@RequestAttribute Client client) {

// ...

}

Multipart

Same in Spring MVC

As explained in Multipart data, ServerWebExchange provides access to multipart content. The best way to handle a file upload form (e.g. from a browser) in a controller is through data binding to a command object:

class MyForm {

private String name;

private MultipartFile file;

// ...

}

@Controller

public class FileUploadController {

@PostMapping("/form")

public String handleFormUpload(MyForm form, BindingResult errors) {

// ...

}

}

Multipart requests can also be submitted from non-browser clients in a RESTful service scenario. For example a file along with JSON:

POST /someUrl

Content-Type: multipart/mixed

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp

Content-Disposition: form-data; name="meta-data"

Content-Type: application/json; charset=UTF-8

Content-Transfer-Encoding: 8bit

{

"name": "value"

}

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp

Content-Disposition: form-data; name="file-data"; filename="file.properties"

Content-Type: text/xml

Content-Transfer-Encoding: 8bit

... File Data ...

You can access individual parts with @RequestPart:

@PostMapping("/")

public String handle(@RequestPart("meta-data") Part metadata,

@RequestPart("file-data") FilePart file) {

// ...

}

To deserialize the raw part content, for example to JSON (similar to @RequestBody), simply declare a concrete target Object, instead of Part:

@PostMapping("/")

public String handle(@RequestPart("meta-data") MetaData metadata) {

// ...

}

@RequestPart can be used in combination with javax.validation.Valid, or Spring’s @Validated annotation, which causes Standard Bean Validation to be applied. By default validation errors cause a WebExchangeBindException which is turned into a 400 (BAD\_REQUEST) response. Alternatively validation errors can be handled locally within the controller through an Errors or BindingResult argument:

@PostMapping("/")

public String handle(@Valid @RequestPart("meta-data") MetaData metadata,

BindingResult result) {

// ...

}

To access all multipart data in as a MultiValueMap use @RequestBody:

@PostMapping("/")

public String handle(@RequestBody Mono<MultiValueMap<String, Part>> parts) {

// ...

}

To access multipart data sequentially, in streaming fashion, use @RequestBody with Flux<Part> instead. For example:

@PostMapping("/")

public String handle(@RequestBody Flux<Part> parts) {

// ...

}

@RequestBody

Same in Spring MVC

Use the @RequestBody annotation to have the request body read and deserialized into an Object through an HttpMessageReader. Below is an example with an @RequestBody argument:

@PostMapping("/accounts")

public void handle(@RequestBody Account account) {

// ...

}

Unlike Spring MVC, in WebFlux the @RequestBody method argument supports reactive types and fully non-blocking reading and (client-to-server) streaming:

@PostMapping("/accounts")

public void handle(@RequestBody Mono<Account> account) {

// ...

}

You can use the HTTP message codecs option of the WebFlux Config to configure or customize message readers.

@RequestBody can be used in combination with javax.validation.Valid, or Spring’s @Validated annotation, which causes Standard Bean Validation to be applied. By default validation errors cause a WebExchangeBindException which is turned into a 400 (BAD\_REQUEST) response. Alternatively validation errors can be handled locally within the controller through an Errors or BindingResult argument:

@PostMapping("/accounts")

public void handle(@Valid @RequestBody Account account, BindingResult result) {

// ...

}

HttpEntity

Same in Spring MVC

HttpEntity is more or less identical to using @RequestBody but based on a container object that exposes request headers and body. Below is an example:

@PostMapping("/accounts")

public void handle(HttpEntity<Account> entity) {

// ...

}

@ResponseBody

Same in Spring MVC

Use the @ResponseBody annotation on a method to have the return serialized to the response body through an HttpMessageWriter. For example:

@GetMapping("/accounts/{id}")

@ResponseBody

public Account handle() {

// ...

}

@ResponseBody is also supported at the class level in which case it is inherited by all controller methods. This is the effect of @RestController which is nothing more than a meta-annotation marked with @Controller and @ResponseBody.

@ResponseBody supports reactive types which means you can return Reactor or RxJava types and have the asynchronous values they produce rendered to the response. For additional details, see HTTP Streaming and JSON rendering.

@ResponseBody methods can be combined with JSON serialization views. See Jackson JSON for details.

You can use the HTTP message codecs option of the WebFlux Config to configure or customize message writing.

ResponseEntity

Same in Spring MVC

ResponseEntity is more or less identical to using @ResponseBody but based on a container object that specifies request headers and body. Below is an example:

@PostMapping("/something")

public ResponseEntity<String> handle() {

// ...

URI location = ...

return new ResponseEntity.created(location).build();

}

Jackson JSON

Jackson serialization views

Same in Spring MVC

Spring WebFlux provides built-in support for Jackson’s Serialization Views which allows rendering only a subset of all fields in an Object. To use it with @ResponseBody or ResponseEntity controller methods, use Jackson’s @JsonView annotation to activate a serialization view class:

@RestController

public class UserController {

@GetMapping("/user")

@JsonView(User.WithoutPasswordView.class)

public User getUser() {

return new User("eric", "7!jd#h23");

}

}

public class User {

public interface WithoutPasswordView {};

public interface WithPasswordView extends WithoutPasswordView {};

private String username;

private String password;

public User() {

}

public User(String username, String password) {

this.username = username;

this.password = password;

}

@JsonView(WithoutPasswordView.class)

public String getUsername() {

return this.username;

}

@JsonView(WithPasswordView.class)

public String getPassword() {

return this.password;

}

}

@JsonView allows an array of view classes but you can only specify only one per controller method. Use a composite interface if you need to activate multiple views.

1.4.4. Model

Same in Spring MVC

The @ModelAttribute annotation can be used:

On a method argument in @RequestMapping methods to create or access an Object from the model, and to bind it to the request through a WebDataBinder.

As a method-level annotation in @Controller or @ControllerAdvice classes helping to initialize the model prior to any @RequestMapping method invocation.

On a @RequestMapping method to mark its return value is a model attribute.

This section discusses @ModelAttribute methods, or the 2nd from the list above. A controller can have any number of @ModelAttribute methods. All such methods are invoked before @RequestMapping methods in the same controller. A @ModelAttribute method can also be shared across controllers via @ControllerAdvice. See the section on Controller Advice for more details.

@ModelAttribute methods have flexible method signatures. They support many of the same arguments as @RequestMapping methods except for @ModelAttribute itself nor anything related to the request body.

An example @ModelAttribute method:

@ModelAttribute

public void populateModel(@RequestParam String number, Model model) {

model.addAttribute(accountRepository.findAccount(number));

// add more ...

}

To add one attribute only:

@ModelAttribute

public Account addAccount(@RequestParam String number) {

return accountRepository.findAccount(number);

}

When a name is not explicitly specified, a default name is chosen based on the Object type as explained in the Javadoc for Conventions. You can always assign an explicit name by using the overloaded addAttribute method or through the name attribute on @ModelAttribute (for a return value).

Spring WebFlux, unlike Spring MVC, explicitly supports reactive types in the model, e.g. Mono<Account> or io.reactivex.Single<Account>. Such asynchronous model attributes may be transparently resolved (and the model updated) to their actual values at the time of @RequestMapping invocation, providing a @ModelAttribute argument is declared without a wrapper, for example:

@ModelAttribute

public void addAccount(@RequestParam String number) {

Mono<Account> accountMono = accountRepository.findAccount(number);

model.addAttribute("account", accountMono);

}

@PostMapping("/accounts")

public String handle(@ModelAttribute Account account, BindingResult errors) {

// ...

}

In addition any model attributes that have a reactive type wrapper are resolved to their actual values (and the model updated) just prior to view rendering.

@ModelAttribute can also be used as a method-level annotation on @RequestMapping methods in which case the return value of the @RequestMapping method is interpreted as a model attribute. This is typically not required, as it is the default behavior in HTML controllers, unless the return value is a String which would otherwise be interpreted as a view name. @ModelAttribute can also help to customize the model attribute name:

@GetMapping("/accounts/{id}")

@ModelAttribute("myAccount")

public Account handle() {

// ...

return account;

}

1.4.5. DataBinder

Same in Spring MVC

@Controller or @ControllerAdvice classes can have @InitBinder methods in order to initialize instances of WebDataBinder, and those in turn are used to:

Bind request parameters (i.e. form data or query) to a model object.

Convert String-based request values such as request parameters, path variables, headers, cookies, and others, to the target type of controller method arguments.

Format model object values as String values when rendering HTML forms.

@InitBinder methods can register controller-specific java.bean.PropertyEditor, or Spring Converter and Formatter components. In addition, the WebFlux Java config can be used to register Converter and Formatter types in a globally shared FormattingConversionService.

@InitBinder methods support many of the same arguments that a @RequestMapping methods do, except for @ModelAttribute (command object) arguments. Typically they’re are declared with a WebDataBinder argument, for registrations, and a void return value. Below is an example:

@Controller

public class FormController {

@InitBinder

public void initBinder(WebDataBinder binder) {

SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

dateFormat.setLenient(false);

binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));

}

// ...

}

Alternatively when using a Formatter-based setup through a shared FormattingConversionService, you could re-use the same approach and register controller-specific Formatter's:

@Controller

public class FormController {

@InitBinder

protected void initBinder(WebDataBinder binder) {

binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));

}

// ...

}

1.4.6. Exceptions

Same in Spring MVC

@Controller and @ControllerAdvice classes can have @ExceptionHandler methods to handle exceptions from controller methods. For example:

@Controller

public class SimpleController {

// ...

@ExceptionHandler

public ResponseEntity<String> handle(IOException ex) {

// ...

}

}

The exception may match against a top-level exception being propagated (i.e. a direct IOException thrown), or against the immediate cause within a top-level wrapper exception (e.g. an IOException wrapped inside an IllegalStateException).

For matching exception types, preferably declare the target exception as a method argument as shown above. Alternatively, the annotation declaration may narrow the exception types to match. We generally recommend to be as specific as possible in the argument signature and to declare your primary root exception mappings on a @ControllerAdvice prioritized with a corresponding order. See the MVC section for details.

An @ExceptionHandler method in WebFlux supports the same method arguments and return values as an @RequestMapping method, with the exception of request body and @ModelAttribute related method arguments.

Support for @ExceptionHandler methods in Spring WebFlux is provided by the HandlerAdapter for @RequestMapping methods. See Exceptions under the DispatcherHandler section for more details.

REST API exceptions

Same in Spring MVC

A common requirement for REST services is to include error details in the body of the response. The Spring Framework does not automatically do this because the representation of error details in the response body is application specific. However a @RestController may use @ExceptionHandler methods with a ResponseEntity return value to set the status and the body of the response. Such methods may also be declared in @ControllerAdvice classes to apply them globally.

Note that Spring WebFlux does not have an equivalent for the Spring MVC ResponseEntityExceptionHandler because WebFlux only raises ResponseStatusException (or subclasses thereof), which and those do not need to be translated translation to an HTTP status code.

1.4.7. Controller Advice

Same in Spring MVC

Typically @ExceptionHandler, @InitBinder, and @ModelAttribute methods apply within the @Controller class (or class hierarchy) they are declared in. If you want such methods to apply more globally, across controllers, you can declare them in a class marked with @ControllerAdvice or @RestControllerAdvice.

@ControllerAdvice is marked with @Component which means such classes can be registered as Spring beans via component scanning. @RestControllerAdvice is also a meta-annotation marked with both @ControllerAdvice and @ResponseBody which essentially means @ExceptionHandler methods are rendered to the response body via message conversion (vs view resolution/template rendering).

On startup, the infrastructure classes for @RequestMapping and @ExceptionHandler methods detect Spring beans of type @ControllerAdvice, and then apply their methods at runtime. Global @ExceptionHandler methods (from an @ControllerAdvice) are applied after local ones (from the @Controller). By contrast global @ModelAttribute and @InitBinder methods are applied before local ones.

By default @ControllerAdvice methods apply to every request, i.e. all controllers, but you can narrow that down to a subset of controllers via attributes on the annotation:

// Target all Controllers annotated with @RestController

@ControllerAdvice(annotations = RestController.class)

public class ExampleAdvice1 {}

// Target all Controllers within specific packages

@ControllerAdvice("org.example.controllers")

public class ExampleAdvice2 {}

// Target all Controllers assignable to specific classes

@ControllerAdvice(assignableTypes = {ControllerInterface.class, AbstractController.class})

public class ExampleAdvice3 {}

Keep in mind the above selectors are evaluated at runtime and may negatively impact performance if used extensively. See the @ControllerAdvice Javadoc for more details.

1.5. Functional Endpoints

Spring WebFlux includes a lightweight, functional programming model in which functions are used to route and handle requests and contracts are designed for immutability. It is an alternative to the annotated-based programming model but otherwise running on the same Reactive Core foundation

1.5.1. Overview

An HTTP request is handled with a HandlerFunction that takes ServerRequest and returns Mono<ServerResponse>, both of which are immutable contracts that offer JDK-8 friendly access to the HTTP request and response. HandlerFunction is the equivalent of an @RequestMapping method in the annotation-based programming model.

Requests are routed to a HandlerFunction with a RouterFunction that takes ServerRequest and returns Mono<HandlerFunction>. When a request is matched to a particular route, the HandlerFunction mapped to the route is used. RouterFunction is the equivalent of an @RequestMapping annotation.

RouterFunctions.route(RequestPredicate, HandlerFunction) provides a router function default implementation that can be used with a number of built-in request predicates. For example:

import static org.springframework.http.MediaType.APPLICATION\_JSON;

import static org.springframework.web.reactive.function.server.RequestPredicates.\*;

import static org.springframework.web.reactive.function.server.RouterFunctions.route;

PersonRepository repository = ...

PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> route =

route(GET("/person/{id}").and(accept(APPLICATION\_JSON)), handler::getPerson)

.andRoute(GET("/person").and(accept(APPLICATION\_JSON)), handler::listPeople)

.andRoute(POST("/person"), handler::createPerson);

public class PersonHandler {

// ...

public Mono<ServerResponse> listPeople(ServerRequest request) {

// ...

}

public Mono<ServerResponse> createPerson(ServerRequest request) {

// ...

}

public Mono<ServerResponse> getPerson(ServerRequest request) {

// ...

}

}

One way to run a RouterFunction is to turn it into an HttpHandler and install it through one of the built-in server adapters:

RouterFunctions.toHttpHandler(RouterFunction)

RouterFunctions.toHttpHandler(RouterFunction, HandlerStrategies)

Most applications will run through the WebFlux Java config, see Running a server.

1.5.2. HandlerFunction

ServerRequest and ServerResponse are immutable interfaces that offer JDK-8 friendly access to the HTTP request and response with Reactive Streams back pressure against the request and response body stream. The request body is represented with a Reactor Flux or Mono. The response body is represented with any Reactive Streams Publisher, including Flux and Mono. For more on that see Reactive Libraries.

ServerRequest

ServerRequest provides access to the HTTP method, URI, headers, and query parameters while access to the body is provided through the body methods.

To extract the request body to a Mono<String>:

Mono<String> string = request.bodyToMono(String.class);

To extract the body to a Flux<Person>, where Person objects are decoded from some serialized form, such as JSON or XML:

Flux<Person> people = request.bodyToFlux(Person.class);

The above are shortcuts that use the more general ServerRequest.body(BodyExtractor) which accepts the BodyExtractor functional, strategy interface. The utility class BodyExtractors provides access to a number of instances. For example, the above can also be written as follows:

Mono<String> string = request.body(BodyExtractors.toMono(String.class));

Flux<Person> people = request.body(BodyExtractors.toFlux(Person.class));

To access form data:

Mono<MultiValueMap<String, String> map = request.body(BodyExtractors.toFormData());

To access multipart data as a map:

Mono<MultiValueMap<String, Part> map = request.body(BodyExtractors.toMultipartData());

To access multiparts, one at a time, in streaming fashion:

Flux<Part> parts = request.body(BodyExtractos.toParts());

ServerResponse

ServerResponse provides access to the HTTP response and since it is immutable, you use a build to create it. The builder can be used to set the response status, to add response headers, or to provide a body. Below is an example with a 200 (OK) response with JSON content:

Mono<Person> person = ...

ServerResponse.ok().contentType(MediaType.APPLICATION\_JSON).body(person, Person.class);

This is how to build a 201 (CREATED) response with "Location" header, and no body:

URI location = ...

ServerResponse.created(location).build();

Handler Classes

We can write a handler function as a lambda. For example:

HandlerFunction<ServerResponse> helloWorld =

request -> ServerResponse.ok().body(fromObject("Hello World"));

That is convenient but in an application we need multiple functions and useful to group related handler functions together into a handler (like an @Controller). For example, here is a class that exposes a reactive Person repository:

import static org.springframework.http.MediaType.APPLICATION\_JSON;

import static org.springframework.web.reactive.function.ServerResponse.ok;

import static org.springframework.web.reactive.function.BodyInserters.fromObject;

public class PersonHandler {

private final PersonRepository repository;

public PersonHandler(PersonRepository repository) {

this.repository = repository;

}

public Mono<ServerResponse> listPeople(ServerRequest request) {

Flux<Person> people = repository.allPeople();

return ok().contentType(APPLICATION\_JSON).body(people, Person.class);

}

public Mono<ServerResponse> createPerson(ServerRequest request) {

Mono<Person> person = request.bodyToMono(Person.class);

return ok().build(repository.savePerson(person));

}

public Mono<ServerResponse> getPerson(ServerRequest request) {

int personId = Integer.valueOf(request.pathVariable("id"));

return repository.getPerson(personId)

.flatMap(person -> ok().contentType(APPLICATION\_JSON).body(fromObject(person)))

.switchIfEmpty(ServerResponse.notFound().build());

}

}

listPeople is a handler function that returns all Person objects found in the repository as JSON.

createPerson is a handler function that stores a new Person contained in the request body. Note that PersonRepository.savePerson(Person) returns Mono<Void>: an empty Mono that emits a completion signal when the person has been read from the request and stored. So we use the build(Publisher<Void>) method to send a response when that completion signal is received, i.e. when the Person has been saved.

getPerson is a handler function that returns a single person, identified via the path variable id. We retrieve that Person via the repository, and create a JSON response if it is found. If it is not found, we use switchIfEmpty(Mono<T>) to return a 404 Not Found response.

1.5.3. RouterFunction

RouterFunction is used to route requests to a HandlerFunction. Typically, you do not write router functions yourself, but rather use RouterFunctions.route(RequestPredicate, HandlerFunction). If the predicate applies, the request is routed to the given HandlerFunction, or otherwise no routing is performed, and that would translate to a 404 (Not Found) response.

Predicates

You can write your own RequestPredicate, but the RequestPredicates utility class offers commonly implementations, based on the request path, HTTP method, content-type, and so on. For example:

RouterFunction<ServerResponse> route =

RouterFunctions.route(RequestPredicates.path("/hello-world"),

request -> Response.ok().body(fromObject("Hello World")));

You can compose multiple request predicates together via:

RequestPredicate.and(RequestPredicate) — both must match.

RequestPredicate.or(RequestPredicate) — either may match.

Many of the predicates from RequestPredicates are composed. For example RequestPredicates.GET(String) is composed from RequestPredicates.method(HttpMethod) and RequestPredicates.path(String).

You can compose multiple router functions into one, such that they’re evaluated in order, and if the first route doesn’t match, the second is evaluated. You can declare more specific routes before more general ones.

Routes

You can compose multiple router functions together via:

RouterFunction.and(RouterFunction)

RouterFunction.andRoute(RequestPredicate, HandlerFunction) — shortcut for RouterFunction.and() with nested RouterFunctions.route().

Using composed routes and predicates, we can then declare the following routes, referring to methods in the PersonHandler, shown in [webflux-fn-handler-class], through method-references:

import static org.springframework.http.MediaType.APPLICATION\_JSON;

import static org.springframework.web.reactive.function.server.RequestPredicates.\*;

PersonRepository repository = ...

PersonHandler handler = new PersonHandler(repository);

RouterFunction<ServerResponse> personRoute =

route(GET("/person/{id}").and(accept(APPLICATION\_JSON)), handler::getPerson)

.andRoute(GET("/person").and(accept(APPLICATION\_JSON)), handler::listPeople)

.andRoute(POST("/person"), handler::createPerson);

1.5.4. Running a server

How do you run a router function in an HTTP server? A simple option is to convert a router function to an HttpHandler using one of the following:

RouterFunctions.toHttpHandler(RouterFunction)

RouterFunctions.toHttpHandler(RouterFunction, HandlerStrategies)

The returned HttpHandler can then be used with a number of servers adapters by following HttpHandler for server-specific instructions.

A more advanced option is to run with a DispatcherHandler-based setup through the WebFlux Config which uses Spring configuration to declare the components quired to process requests. The WebFlux Java config declares the following infrastructure components to support functional endpoints:

RouterFunctionMapping — detects one or more RouterFunction<?> beans in the Spring configuration, combines them via RouterFunction.andOther, and routes requests to the resulting composed RouterFunction.

HandlerFunctionAdapter — simple adapter that allows the DispatcherHandler to invoke a HandlerFunction that was mapped to a request.

ServerResponseResultHandler — handles the result from the invocation of a HandlerFunction by invoking the writeTo method of the ServerResponse.

The above components allow functional endpoints to fit within the DispatcherHandler request processing lifecycle, and also potentially run side by side with annotated controllers, if any are declared. It is also how functional endpoints are enabled the Spring Boot WebFlux starter.

Below is example WebFlux Java config (see DispatcherHandler for how to run):

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Bean

public RouterFunction<?> routerFunctionA() {

// ...

}

@Bean

public RouterFunction<?> routerFunctionB() {

// ...

}

// ...

@Override

public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {

// configure message conversion...

}

@Override

public void addCorsMappings(CorsRegistry registry) {

// configure CORS...

}

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

// configure view resolution for HTML rendering...

}

}

1.5.5. HandlerFilterFunction

Routes mapped by a router function can be filtered by calling RouterFunction.filter(HandlerFilterFunction), where HandlerFilterFunction is essentially a function that takes a ServerRequest and HandlerFunction, and returns a ServerResponse. The handler function parameter represents the next element in the chain: this is typically the HandlerFunction that is routed to, but can also be another FilterFunction if multiple filters are applied. With annotations, similar functionality can be achieved using @ControllerAdvice and/or a ServletFilter. Let’s add a simple security filter to our route, assuming that we have a SecurityManager that can determine whether a particular path is allowed:

import static org.springframework.http.HttpStatus.UNAUTHORIZED;

SecurityManager securityManager = ...

RouterFunction<ServerResponse> route = ...

RouterFunction<ServerResponse> filteredRoute =

route.filter((request, next) -> {

if (securityManager.allowAccessTo(request.path())) {

return next.handle(request);

}

else {

return ServerResponse.status(UNAUTHORIZED).build();

}

});

You can see in this example that invoking the next.handle(ServerRequest) is optional: we only allow the handler function to be executed when access is allowed.

CORS support for functional endpoints is provided via a dedicated CorsWebFilter.

1.6. URI Links

Same in Spring MVC

This section describes various options available in the Spring Framework to prepare URIs.

1.6.1. UriComponents

Spring MVC and Spring WebFlux

UriComponentsBuilder helps to build URI’s from URI templates with variables:

UriComponents uriComponents = UriComponentsBuilder

.fromUriString("http://example.com/hotels/{hotel}")

.queryParam("q", "{q}")

.encode()

.build();

URI uri = uriComponents.expand("Westin", "123").toUri();

Static factory method with a URI template.

Add and/or replace URI components.

Request to have the URI template and URI variables encoded.

Build a UriComponents.

Expand variables, and obtain the URI.

The above can be consolidated into one chain and shortened with buildAndExpand:

URI uri = UriComponentsBuilder

.fromUriString("http://example.com/hotels/{hotel}")

.queryParam("q", "{q}")

.encode()

.buildAndExpand("Westin", "123")

.toUri();

It can be shortened further by going directly to URI (which implies encoding):

URI uri = UriComponentsBuilder

.fromUriString("http://example.com/hotels/{hotel}")

.queryParam("q", "{q}")

.build("Westin", "123");

Or shorter further yet, with a full URI template:

URI uri = UriComponentsBuilder

.fromUriString("http://example.com/hotels/{hotel}?q={q}")

.build("Westin", "123");

1.6.2. UriBuilder

Spring MVC and Spring WebFlux

UriComponentsBuilder implements UriBuilder. A UriBuilder in turn can be created with a UriBuilderFactory. Together UriBuilderFactory and UriBuilder provide a pluggable mechanism to build URIs from URI templates, based on shared configuration such as a base url, encoding preferences, and others.

The RestTemplate and the WebClient can be configured with a UriBuilderFactory to customize the preparation of URIs. DefaultUriBuilderFactory is a default implementation of UriBuilderFactory that uses UriComponentsBuilder internally and exposes shared configuration options.

RestTemplate example:

// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "http://example.org";

DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);

factory.setEncodingMode(EncodingMode.TEMPLATE\_AND\_VARIABLES);

RestTemplate restTemplate = new RestTemplate();

restTemplate.setUriTemplateHandler(factory);

WebClient example:

// import org.springframework.web.util.DefaultUriBuilderFactory.EncodingMode;

String baseUrl = "http://example.org";

DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl);

factory.setEncodingMode(EncodingMode.TEMPLATE\_AND\_VARIABLES);

WebClient client = WebClient.builder().uriBuilderFactory(factory).build();

In addition DefaultUriBuilderFactory can also be used directly. It is similar to using UriComponentsBuilder but instead of static factory methods, it is an actual instance that holds configuration and preferences:

String baseUrl = "http://example.com";

DefaultUriBuilderFactory uriBuilderFactory = new DefaultUriBuilderFactory(baseUrl);

URI uri = uriBuilderFactory.uriString("/hotels/{hotel}")

.queryParam("q", "{q}")

.build("Westin", "123");

1.6.3. URI Encoding

Spring MVC and Spring WebFlux

UriComponentsBuilder exposes encoding options at 2 levels:

UriComponentsBuilder#encode() - pre-encodes the URI template first, then strictly encodes URI variables when expanded.

UriComponents#encode() - encodes URI components after URI variables are expanded.

Both options replace non-ASCII and illegal characters with escaped octets, however option 1 also replaces characters with reserved meaning that appear in URI variables.

Consider ";" which is legal in a path but has reserved meaning. Option 1 replaces ";" with "%3B" in URI variables but not in the URI template. By contrast, option 2 never replaces ";" since it is a legal character in a path.

For most cases option 1 is likely to give the expected result because it treats URI variables as opaque data to be fully encoded, while option 2 is useful only if URI variables intentionally contain reserved characters.

Example usage using option 1:

URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")

.queryParam("q", "{q}")

.encode()

.buildAndExpand("New York", "foo+bar")

.toUri();

// Result is "/hotel%20list/New%20York?foo%2Bbar"

The above can be shortened by going directly to URI (which implies encoding):

URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}")

.queryParam("q", "{q}")

.build("New York", "foo+bar")

Or shorter further yet, with a full URI template:

URI uri = UriComponentsBuilder.fromPath("/hotel list/{city}?q={q}")

.build("New York", "foo+bar")

The WebClient and the RestTemplate expand and encode URI templates internally through the UriBuilderFactory strategy. Both can be configured with a custom strategy:

String baseUrl = "http://example.com";

DefaultUriBuilderFactory factory = new DefaultUriBuilderFactory(baseUrl)

factory.setEncodingMode(EncodingMode.TEMPLATE\_AND\_VALUES);

// Customize the RestTemplate..

RestTemplate restTemplate = new RestTemplate();

restTemplate.setUriTemplateHandler(factory);

// Customize the WebClient..

WebClient client = WebClient.builder().uriBuilderFactory(factory).build();

The DefaultUriBuilderFactory implementation uses UriComponentsBuilder internally to expand and encode URI templates. As a factory it provides a single place to configure the approach to encoding based on one of the below encoding modes:

TEMPLATE\_AND\_VALUES — uses UriComponentsBuilder#encode(), corresponding to option 1 above, to pre-encode the URI template and strictly encode URI variables when expanded.

VALUES\_ONLY — does not encode the URI template and instead applies strict encoding to URI variables via UriUtils#encodeUriUriVariables prior to expanding them into the template.

URI\_COMPONENTS — uses UriComponents#encode(), corresponding to option 2 above, to encode URI component value after URI variables are expanded.

NONE — no encoding is applied.

Out of the box the RestTemplate is set to EncodingMode.URI\_COMPONENTS for historic reasons and for backwards compatibility. The WebClient relies on the default value in DefaultUriBuilderFactory which was changed from EncodingMode.URI\_COMPONENTS in 5.0.x to EncodingMode.TEMPLATE\_AND\_VALUES in 5.1.

1.7. CORS

Same in Spring MVC

1.7.1. Introduction

Same in Spring MVC

For security reasons browsers prohibit AJAX calls to resources outside the current origin. For example you could have your bank account in one tab and evil.com in another. Scripts from evil.com should not be able to make AJAX requests to your bank API with your credentials, e.g. withdrawing money from your account!

Cross-Origin Resource Sharing (CORS) is a W3C specification implemented by most browsers that allows you to specify what kind of cross domain requests are authorized rather than using less secure and less powerful workarounds based on IFRAME or JSONP.

1.7.2. Processing

Same in Spring MVC

The CORS specification distinguishes between preflight, simple, and actual requests. To learn how CORS works, you can read this article, among many others, or refer to the specification for more details.

Spring WebFlux HandlerMapping's provide built-in support for CORS. After successfully mapping a request to a handler, HandlerMapping's check the CORS configuration for the given request and handler and take further actions. Preflight requests are handled directly while simple and actual CORS requests are intercepted, validated, and have required CORS response headers set.

In order to enable cross-origin requests (i.e. the Origin header is present and differs from the host of the request) you need to have some explicitly declared CORS configuration. If no matching CORS configuration is found, preflight requests are rejected. No CORS headers are added to the responses of simple and actual CORS requests and consequently browsers reject them.

Each HandlerMapping can be configured individually with URL pattern based CorsConfiguration mappings. In most cases applications will use the WebFlux Java config to declare such mappings, which results in a single, global map passed to all HadlerMappping's.

Global CORS configuration at the HandlerMapping level can be combined with more fine-grained, handler-level CORS configuration. For example annotated controllers can use class or method-level @CrossOrigin annotations (other handlers can implement CorsConfigurationSource).

The rules for combining global and local configuration are generally additive — e.g. all global and all local origins. For those attributes where only a single value can be accepted such as allowCredentials and maxAge, the local overrides the global value. See CorsConfiguration#combine(CorsConfiguration) for more details.

To learn more from the source or make advanced customizations, check:

CorsConfiguration

CorsProcessor, DefaultCorsProcessor

AbstractHandlerMapping

1.7.3. @CrossOrigin

Same in Spring MVC

The @CrossOrigin annotation enables cross-origin requests on annotated controller methods:

@RestController

@RequestMapping("/account")

public class AccountController {

@CrossOrigin

@GetMapping("/{id}")

public Mono<Account> retrieve(@PathVariable Long id) {

// ...

}

@DeleteMapping("/{id}")

public Mono<Void> remove(@PathVariable Long id) {

// ...

}

}

By default @CrossOrigin allows:

All origins.

All headers.

All HTTP methods to which the controller method is mapped.

allowedCredentials is not enabled by default since that establishes a trust level that exposes sensitive user-specific information such as cookies and CSRF tokens, and should only be used where appropriate.

maxAge is set to 30 minutes.

@CrossOrigin is supported at the class level too and inherited by all methods:

@CrossOrigin(origins = "http://domain2.com", maxAge = 3600)

@RestController

@RequestMapping("/account")

public class AccountController {

@GetMapping("/{id}")

public Mono<Account> retrieve(@PathVariable Long id) {

// ...

}

@DeleteMapping("/{id}")

public Mono<Void> remove(@PathVariable Long id) {

// ...

}

}

CrossOrigin can be used at both class and method-level:

@CrossOrigin(maxAge = 3600)

@RestController

@RequestMapping("/account")

public class AccountController {

@CrossOrigin("http://domain2.com")

@GetMapping("/{id}")

public Mono<Account> retrieve(@PathVariable Long id) {

// ...

}

@DeleteMapping("/{id}")

public Mono<Void> remove(@PathVariable Long id) {

// ...

}

}

1.7.4. Global Config

Same in Spring MVC

In addition to fine-grained, controller method level configuration you’ll probably want to define some global CORS configuration too. You can set URL-based CorsConfiguration mappings individually on any HandlerMapping. Most applications however will use the WebFlux Java config to do that.

By default global configuration enables the following:

All origins.

All headers.

GET, HEAD, and POST methods.

allowedCredentials is not enabled by default since that establishes a trust level that exposes sensitive user-specific information such as cookies and CSRF tokens, and should only be used where appropriate.

maxAge is set to 30 minutes.

To enable CORS in the WebFlux Java config, use the CorsRegistry callback:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/api/\*\*")

.allowedOrigins("http://domain2.com")

.allowedMethods("PUT", "DELETE")

.allowedHeaders("header1", "header2", "header3")

.exposedHeaders("header1", "header2")

.allowCredentials(true).maxAge(3600);

// Add more mappings...

}

}

1.7.5. CORS WebFilter

Same in Spring MVC

You can apply CORS support through the built-in CorsWebFilter, which is a good fit with functional endpoints.

To configure the filter, you can declare a CorsWebFilter bean and pass a CorsConfigurationSource to its constructor:

@Bean

CorsWebFilter corsFilter() {

CorsConfiguration config = new CorsConfiguration();

// Possibly...

// config.applyPermitDefaultValues()

config.setAllowCredentials(true);

config.addAllowedOrigin("http://domain1.com");

config.addAllowedHeader("");

config.addAllowedMethod("");

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

source.registerCorsConfiguration("/\*\*", config);

return new CorsWebFilter(source);

}

1.8. Web Security

Same in Spring MVC

The Spring Security project provides support for protecting web applications from malicious exploits. Check out the Spring Security reference documentation including:

WebFlux Security

"WebFlux Testing Support"

CSRF Protection

Security Response Headers

1.9. View Technologies

Same in Spring MVC

The use of view technologies in Spring WebFlux is pluggable, whether you decide to use Thymeleaf, FreeMarker, or other, is primarily a matter of a configuration change. This chapter covers view technologies integrated with Spring WebFlux. We assume you are already familiar with View Resolution.

1.9.1. Thymeleaf

Same in Spring MVC

Thymeleaf is modern server-side Java template engine that emphasizes natural HTML templates that can be previewed in a browser by double-clicking, which is very helpful for independent work on UI templates, e.g. by designer, without the need for a running server. Thymeleaf offers an extensive set of features and it is actively developed and maintained. For a more complete introduction see the Thymeleaf project home page.

The Thymeleaf integration with Spring WebFlux is managed by the Thymeleaf project. The configuration involves a few bean declarations such as SpringResourceTemplateResolver, SpringWebFluxTemplateEngine, and ThymeleafReactiveViewResolver. For more details see Thymeleaf+Spring and the WebFlux integration announcement.

1.9.2. FreeMarker

Same in Spring MVC

Apache FreeMarker is a template engine for generating any kind of text output from HTML to email, and others. The Spring Framework has a built-in integration for using Spring WebFlux with FreeMarker templates.

View config

Same in Spring MVC

To configure FreeMarker as a view technology:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.freemarker();

}

// Configure FreeMarker...

@Bean

public FreeMarkerConfigurer freeMarkerConfigurer() {

FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();

configurer.setTemplateLoaderPath("classpath:/templates");

return configurer;

}

}

Your templates need to be stored in the directory specified by the FreeMarkerConfigurer shown above. Given the above configuration if your controller returns the view name "welcome" then the resolver will look for the classpath:/templates/freemarker/welcome.ftl template.

FreeMarker config

Same in Spring MVC

FreeMarker 'Settings' and 'SharedVariables' can be passed directly to the FreeMarker Configuration object managed by Spring by setting the appropriate bean properties on the FreeMarkerConfigurer bean. The freemarkerSettings property requires a java.util.Properties object and the freemarkerVariables property requires a java.util.Map.

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

// ...

@Bean

public FreeMarkerConfigurer freeMarkerConfigurer() {

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

variables.put("xml\_escape", new XmlEscape());

FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();

configurer.setTemplateLoaderPath("classpath:/templates");

configurer.setFreemarkerVariables(variables);

return configurer;

}

}

See the FreeMarker documentation for details of settings and variables as they apply to the Configuration object.

1.9.3. Script Views

Same in Spring MVC

The Spring Framework has a built-in integration for using Spring WebFlux with any templating library that can run on top of the JSR-223 Java scripting engine. Below is a list of templating libraries we’ve tested on different script engines:

Handlebars

Nashorn

Mustache

Nashorn

React

Nashorn

EJS

Nashorn

ERB

JRuby

String templates

Jython

Kotlin Script templating

Kotlin

The basic rule for integrating any other script engine is that it must implement the ScriptEngine and Invocable interfaces.

Requirements

Same in Spring MVC

You need to have the script engine on your classpath:

Nashorn JavaScript engine is provided with Java 8+. Using the latest update release available is highly recommended.

JRuby should be added as a dependency for Ruby support.

Jython should be added as a dependency for Python support.

org.jetbrains.kotlin:kotlin-script-util dependency and a META-INF/services/javax.script.ScriptEngineFactory file containing a org.jetbrains.kotlin.script.jsr223.KotlinJsr223JvmLocalScriptEngineFactory line should be added for Kotlin script support, see this example for more details.

You need to have the script templating library. One way to do that for Javascript is through WebJars.

Script templates

Same in Spring MVC

Declare a ScriptTemplateConfigurer bean in order to specify the script engine to use, the script files to load, what function to call to render templates, and so on. Below is an example with Mustache templates and the Nashorn JavaScript engine:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.scriptTemplate();

}

@Bean

public ScriptTemplateConfigurer configurer() {

ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();

configurer.setEngineName("nashorn");

configurer.setScripts("mustache.js");

configurer.setRenderObject("Mustache");

configurer.setRenderFunction("render");

return configurer;

}

}

The render function is called with the following parameters:

String template: the template content

Map model: the view model

RenderingContext renderingContext: the RenderingContext that gives access to the application context, the locale, the template loader and the url (since 5.0)

Mustache.render() is natively compatible with this signature, so you can call it directly.

If your templating technology requires some customization, you may provide a script that implements a custom render function. For example, Handlerbars needs to compile templates before using them, and requires a polyfill in order to emulate some browser facilities not available in the server-side script engine.

@Configuration

@EnableWebMvc

public class WebConfig implements WebFluxConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.scriptTemplate();

}

@Bean

public ScriptTemplateConfigurer configurer() {

ScriptTemplateConfigurer configurer = new ScriptTemplateConfigurer();

configurer.setEngineName("nashorn");

configurer.setScripts("polyfill.js", "handlebars.js", "render.js");

configurer.setRenderFunction("render");

configurer.setSharedEngine(false);

return configurer;

}

}

Setting the sharedEngine property to false is required when using non thread-safe script engines with templating libraries not designed for concurrency, like Handlebars or React running on Nashorn for example. In that case, Java 8u60 or greater is required due to this bug.

polyfill.js only defines the window object needed by Handlebars to run properly:

var window = {};

This basic render.js implementation compiles the template before using it. A production ready implementation should also store and reused cached templates / pre-compiled templates. This can be done on the script side, as well as any customization you need (managing template engine configuration for example).

function render(template, model) {

var compiledTemplate = Handlebars.compile(template);

return compiledTemplate(model);

}

Check out the Spring Framework unit tests, java, and resources, for more configuration examples.

1.9.4. JSON, XML

Same in Spring MVC

For Content negotiation purposes it is useful to be able to alternate between rendering a model with an HTML template or as other formats such as JSON or XML, depending on the content type requested by the client. To support this Spring WebFlux provides the HttpMessageWriterView that can be used to plug in any of the available Codecs from spring-web such as Jackson2JsonEncoder, Jackson2SmileEncoder, or Jaxb2XmlEncoder.

Unlike other view technologies, HttpMessageWriterView does not require a ViewResolver, but instead is configured as a default view. You can configure one more such default views, wrapping different HttpMessageWriter's or Encoder's. The one that matches the requested content type is used at runtime.

In most cases a model will contain multiple attributes. In order to determine which one to serialize, HttpMessageWriterView can be configured with the name of the model attribute to use render, of if the model contains only one attribute, it will be used.

1.10. HTTP Caching

Same in Spring MVC

HTTP caching can significantly improve the performance of a web application. HTTP caching revolves around the "Cache-Control" response header and subsequently conditional request headers such as "Last-Modified" and "ETag". "Cache-Control" advises private (e.g. browser) and public (e.g. proxy) caches how to cache and re-use responses. An "ETag" header is used to make a conditional request that may result in a 304 (NOT\_MODIFIED) without a body, if the content has not changed. "ETag" can be seen as a more sophisticated successor to the Last-Modified header.

This section describes HTTP caching related options available in Spring Web MVC.

1.10.1. CacheControl

Same in Spring MVC

CacheControl provides support for configuring settings related to the "Cache-Control" header and is accepted as an argument in a number of places:

Controllers

Static resources

While RFC 7234 describes all possible directives for the "Cache-Control" response header, the CacheControl type takes a use case oriented approach focusing on the common scenarios:

// Cache for an hour - "Cache-Control: max-age=3600"

CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

// Prevent caching - "Cache-Control: no-store"

CacheControl ccNoStore = CacheControl.noStore();

// Cache for ten days in public and private caches,

// public caches should not transform the response

// "Cache-Control: max-age=864000, public, no-transform"

CacheControl ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS).noTransform().cachePublic();

1.10.2. Controllers

Same in Spring MVC

Controllers can add explicit support for HTTP caching. This is recommended since the lastModified or ETag value for a resource needs to be calculated before it can be compared against conditional request headers. A controller can add an ETag and "Cache-Control" settings to a ResponseEntity:

@GetMapping("/book/{id}")

public ResponseEntity<Book> showBook(@PathVariable Long id) {

Book book = findBook(id);

String version = book.getVersion();

return ResponseEntity

.ok()

.cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))

.eTag(version) // lastModified is also available

.body(book);

}

This will send an 304 (NOT\_MODIFIED) response with an empty body, if the comparison to the conditional request headers indicates the content has not changed. Otherwise the "ETag" and "Cache-Control" headers will be added to the response.

The check against conditional request headers can also be made in the controller:

@RequestMapping

public String myHandleMethod(ServerWebExchange exchange, Model model) {

long eTag = ...

if (exchange.checkNotModified(eTag)) {

return null;

}

model.addAttribute(...);

return "myViewName";

}

Application-specific calculation.

Response has been set to 304 (NOT\_MODIFIED), no further processing.

Continue with request processing.

There are 3 variants for checking conditional requests against eTag values, lastModified values, or both. For conditional "GET" and "HEAD" requests, the response may be set to 304 (NOT\_MODIFIED). For conditional "POST", "PUT", and "DELETE", the response would be set to 409 (PRECONDITION\_FAILED) instead to prevent concurrent modification.

1.10.3. Static resources

Same in Spring MVC

Static resources should be served with a "Cache-Control" and conditional response headers for optimal performance. See section on configuring Static resources.

1.11. WebFlux Config

Same in Spring MVC

The WebFlux Java config declares components required to process requests with annotated controllers or functional endpoints, and it offers an API to customize the configuration. That means you do not need to understand the underlying beans created by the Java config but, if you want to, it’s very easy to see them in WebFluxConfigurationSupport or read more what they are in Special bean types.

For more advanced customizations, not available in the configuration API, it is also possible to gain full control over the configuration through the Advanced config mode.

1.11.1. Enable WebFlux config

Same in Spring MVC

Use the @EnableWebFlux annotation in your Java config:

@Configuration

@EnableWebFlux

public class WebConfig {

}

The above registers a number of Spring WebFlux infrastructure beans also adapting to dependencies available on the classpath — for JSON, XML, etc.

1.11.2. WebFlux config API

Same in Spring MVC

In your Java config implement the WebFluxConfigurer interface:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

// Implement configuration methods...

}

1.11.3. Conversion, formatting

Same in Spring MVC

By default formatters for Number and Date types are installed, including support for the @NumberFormat and @DateTimeFormat annotations. Full support for the Joda-Time formatting library is also installed if Joda-Time is present on the classpath.

To register custom formatters and converters:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void addFormatters(FormatterRegistry registry) {

// ...

}

}

See FormatterRegistrar SPI and the FormattingConversionServiceFactoryBean for more information on when to use FormatterRegistrars.

1.11.4. Validation

Same in Spring MVC

By default if Bean Validation is present on the classpath — e.g. Hibernate Validator, the LocalValidatorFactoryBean is registered as a global Validator for use with @Valid and Validated on @Controller method arguments.

In your Java config, you can customize the global Validator instance:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public Validator getValidator(); {

// ...

}

}

Note that you can also register Validator's locally:

@Controller

public class MyController {

@InitBinder

protected void initBinder(WebDataBinder binder) {

binder.addValidators(new FooValidator());

}

}

If you need to have a LocalValidatorFactoryBean injected somewhere, create a bean and mark it with @Primary in order to avoid conflict with the one declared in the MVC config.

1.11.5. Content type resolvers

Same in Spring MVC

You can configure how Spring WebFlux determines the requested media types for @Controller's from the request. By default only the "Accept" header is checked but you can also enable a query parameter based strategy.

To customize the requested content type resolution:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void configureContentTypeResolver(RequestedContentTypeResolverBuilder builder) {

// ...

}

}

1.11.6. HTTP message codecs

Same in Spring MVC

To customize how the request and response body are read and written:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void configureHttpMessageCodecs(ServerCodecConfigurer configurer) {

// ...

}

}

ServerCodecConfigurer provides a set of default readers and writers. You can use it to add more readers and writers, customize the default ones, or replace the default ones completely.

For Jackson JSON and XML, consider using the Jackson2ObjectMapperBuilder which customizes Jackson’s default properties with the following ones:

DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES is disabled.

MapperFeature.DEFAULT\_VIEW\_INCLUSION is disabled.

It also automatically registers the following well-known modules if they are detected on the classpath:

jackson-datatype-jdk7: support for Java 7 types like java.nio.file.Path.

jackson-datatype-joda: support for Joda-Time types.

jackson-datatype-jsr310: support for Java 8 Date & Time API types.

jackson-datatype-jdk8: support for other Java 8 types like Optional.

1.11.7. View resolvers

Same in Spring MVC

To configure view resolution:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

// ...

}

}

The ViewResolverRegistry has shortcuts for view technologies that the Spring Framework integrates with. Here is an example with FreeMarker which also requires configuring the underlying FreeMarker view technology:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.freeMarker();

}

// Configure Freemarker...

@Bean

public FreeMarkerConfigurer freeMarkerConfigurer() {

FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();

configurer.setTemplateLoaderPath("classpath:/templates");

return configurer;

}

}

You can also plug in any ViewResolver implementation:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

ViewResolver resolver = ... ;

registry.viewResolver(resolver);

}

}

To support Content negotiation and rendering other formats through view resolution, besides HTML, you can configure one or more default views based on the HttpMessageWriterView implementation which accepts any of the available Codecs from spring-web:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void configureViewResolvers(ViewResolverRegistry registry) {

registry.freeMarker();

Jackson2JsonEncoder encoder = new Jackson2JsonEncoder();

registry.defaultViews(new HttpMessageWriterView(encoder));

}

// ...

}

See View Technologies for more on the view technologies integrated with Spring WebFlux.

1.11.8. Static resources

Same in Spring MVC

This option provides a convenient way to serve static resources from a list of Resource-based locations.

In the example below, given a request that starts with "/resources", the relative path is used to find and serve static resources relative to "/static" on the classpath. Resources will be served with a 1-year future expiration to ensure maximum use of the browser cache and a reduction in HTTP requests made by the browser. The Last-Modified header is also evaluated and if present a 304 status code is returned.

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*")

.addResourceLocations("/public", "classpath:/static/")

.setCacheControl(CacheControl.maxAge(365, TimeUnit.DAYS));

}

}

The resource handler also supports a chain of ResourceResolver's and ResourceTransformer's. which can be used to create a toolchain for working with optimized resources.

The VersionResourceResolver can be used for versioned resource URLs based on an MD5 hash computed from the content, a fixed application version, or other. A ContentVersionStrategy (MD5 hash) is a good choice with some notable exceptions such as JavaScript resources used with a module loader.

For example in your Java config;

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry.addResourceHandler("/resources/\*\*")

.addResourceLocations("/public/")

.resourceChain(true)

.addResolver(new VersionResourceResolver().addContentVersionStrategy("/\*\*"));

}

}

You can use ResourceUrlProvider to rewrite URLs and apply the full chain of resolvers and transformers — e.g. to insert versions. The WebFlux config provides a ResourceUrlProvider so it can be injected into others.

Unlike Spring MVC at present in WebFlux there is no way to transparently rewrite static resource URLs since there are no view technologies that can make use of a non-blocking chain of resolvers and transformers. When serving only local resources the workaround is to use ResourceUrlProvider directly (e.g. through a custom tag) and block.

Note that when using both GzipResourceResolver and VersionedResourceResolver, they must be registered in that order to ensure content based versions are always computed reliably based on the unencoded file.

WebJars is also supported via WebJarsResourceResolver and automatically registered when "org.webjars:webjars-locator" is present on the classpath. The resolver can re-write URLs to include the version of the jar and can also match to incoming URLs without versions — e.g. "/jquery/jquery.min.js" to "/jquery/1.2.0/jquery.min.js".

1.11.9. Path Matching

Same in Spring MVC

Spring WebFlux uses parsed representation of path patterns — i.e. PathPattern, and also the incoming request path — i.e. RequestPath, which eliminates the need to indicate whether to decode the request path, or remove semicolon content, since PathPattern can now access decoded path segment values and match safely.

Spring WebFlux also does not support suffix pattern matching so effectively there are only two minor options to customize related to path matching — whether to match trailing slashes (true by default) and whether the match is case-sensitive (false).

To customize those options:

@Configuration

@EnableWebFlux

public class WebConfig implements WebFluxConfigurer {

@Override

public void configurePathMatch(PathMatchConfigurer configurer) {

// ...

}

}

1.11.10. Advanced config mode

Same in Spring MVC

@EnableWebFlux imports DelegatingWebFluxConfiguration that (1) provides default Spring configuration for WebFlux applications and (2) detects and delegates to WebFluxConfigurer's to customize that configuration.

For advanced mode, remove @EnableWebFlux and extend directly from DelegatingWebFluxConfiguration instead of implementing WebFluxConfigurer:

@Configuration

public class WebConfig extends DelegatingWebFluxConfiguration {

// ...

}

You can keep existing methods in WebConfig but you can now also override bean declarations from the base class and you can still have any number of other WebMvcConfigurer's on the classpath.

1.12. HTTP/2

Same in Spring MVC

Servlet 4 containers are required to support HTTP/2 and Spring Framework 5 is compatible with Servlet API 4. From a programming model perspective there is nothing specific that applications need to do. However there are considerations related to server configuration. For more details please check out the HTTP/2 wiki page.

Currently Spring WebFlux does not support HTTP/2 with Netty. There is also no support for pushing resources programmatically to the client.

2. WebClient

The spring-webflux module includes a reactive, non-blocking client for HTTP requests with a functional-style API client and Reactive Streams support. WebClient depends on a lower level HTTP client library to execute requests and that support is pluggable.

WebClient uses the same codecs as WebFlux server applications do, and shares a common base package, some common APIs, and infrastructure with the server functional web framework. The API exposes Reactor Flux and Mono types, also see Reactive Libraries. By default it uses it uses Reactor Netty as the HTTP client library but others can be plugged in through a custom ClientHttpConnector.

By comparison to the RestTemplate, the WebClient is:

non-blocking, reactive, and supports higher concurrency with less hardware resources.

provides a functional API that takes advantage of Java 8 lambdas.

supports both synchronous and asynchronous scenarios.

supports streaming up or down from a server.

The RestTemplate is not a good fit for use in non-blocking applications, and therefore Spring WebFlux application should always use the WebClient. The WebClient should also be preferred in Spring MVC, in most high concurrency scenarios, and for composing a sequence of remote, inter-dependent calls.

2.1. Retrieve

The retrieve() method is the easiest way to get a response body and decode it:

WebClient client = WebClient.create("http://example.org");

Mono<Person> result = client.get()

.uri("/persons/{id}", id).accept(MediaType.APPLICATION\_JSON)

.retrieve()

.bodyToMono(Person.class);

You can also get a stream of objects decoded from the response:

Flux<Quote> result = client.get()

.uri("/quotes").accept(MediaType.TEXT\_EVENT\_STREAM)

.retrieve()

.bodyToFlux(Quote.class);

By default, responses with 4xx or 5xx status codes result in an error of type WebClientResponseException but you can customize that:

Mono<Person> result = client.get()

.uri("/persons/{id}", id).accept(MediaType.APPLICATION\_JSON)

.retrieve()

.onStatus(HttpStatus::is4xxServerError, response -> ...)

.onStatus(HttpStatus::is5xxServerError, response -> ...)

.bodyToMono(Person.class);

2.2. Exchange

The exchange() method provides more control. The below example is equivalent to retrieve() but also provides access to the ClientResponse:

Mono<Person> result = client.get()

.uri("/persons/{id}", id).accept(MediaType.APPLICATION\_JSON)

.exchange()

.flatMap(response -> response.bodyToMono(Person.class));

At this level you can also create a full ResponseEntity:

Mono<ResponseEntity<Person>> result = client.get()

.uri("/persons/{id}", id).accept(MediaType.APPLICATION\_JSON)

.exchange()

.flatMap(response -> response.toEntity(Person.class));

Note that unlike retrieve(), with exchange() there are no automatic error signals for 4xx and 5xx responses. You have to check the status code and decide how to proceed.

When using exchange() you must always use any of the body or toEntity methods of ClientResponse to ensure resources are released and to avoid potential issues with HTTP connection pooling. You can use bodyToMono(Void.class) if no response content is expected. However keep in mind that if the response does have content, the connection will be closed and will not be placed back in the pool.

2.3. Request body

The request body can be encoded from an Object:

Mono<Person> personMono = ... ;

Mono<Void> result = client.post()

.uri("/persons/{id}", id)

.contentType(MediaType.APPLICATION\_JSON)

.body(personMono, Person.class)

.retrieve()

.bodyToMono(Void.class);

You can also have a stream of objects encoded:

Flux<Person> personFlux = ... ;

Mono<Void> result = client.post()

.uri("/persons/{id}", id)

.contentType(MediaType.APPLICATION\_STREAM\_JSON)

.body(personFlux, Person.class)

.retrieve()

.bodyToMono(Void.class);

Or if you have the actual value, use the syncBody shortcut method:

Person person = ... ;

Mono<Void> result = client.post()

.uri("/persons/{id}", id)

.contentType(MediaType.APPLICATION\_JSON)

.syncBody(person)

.retrieve()

.bodyToMono(Void.class);

2.3.1. Form data

To send form data, provide a MultiValueMap<String, String> as the body. Note that the content is automatically set to "application/x-www-form-urlencoded" by the FormHttpMessageWriter:

MultiValueMap<String, String> formData = ... ;

Mono<Void> result = client.post()

.uri("/path", id)

.syncBody(formData)

.retrieve()

.bodyToMono(Void.class);

You can also supply form data in-line via BodyInserters:

import static org.springframework.web.reactive.function.BodyInserters.\*;

Mono<Void> result = client.post()

.uri("/path", id)

.body(fromFormData("k1", "v1").with("k2", "v2"))

.retrieve()

.bodyToMono(Void.class);

2.3.2. Multipart data

To send multipart data, you need to provide a MultiValueMap<String, ?> whose values are either Objects representing part content, or HttpEntity representing the content and headers for a part. MultipartBodyBuilder provides a convenient API to prepare a multipart request:

MultipartBodyBuilder builder = new MultipartBodyBuilder();

builder.part("fieldPart", "fieldValue");

builder.part("filePart", new FileSystemResource("...logo.png"));

builder.part("jsonPart", new Person("Jason"));

MultiValueMap<String, HttpEntity<?>> parts = builder.build();

In most cases you do not have to specify the Content-Type for each part. The content type is determined automatically based on the HttpMessageWriter chosen to serialize it, or in the case of a Resource based on the file extension. If necessary you can explicitly provide the MediaType to use for each part through one fo the overloaded builder part methods.

Once a MultiValueMap is prepared, the easiest way to pass it to the the WebClient is through the syncBody method:

MultipartBodyBuilder builder = ...;

Mono<Void> result = client.post()

.uri("/path", id)

.syncBody(builder.build())

.retrieve()

.bodyToMono(Void.class);

If the MultiValueMap contains at least one non-String value, which could also be represent regular form data (i.e. "application/x-www-form-urlencoded"), you don’t have to set the Content-Type to "multipart/form-data". This is always the case when using MultipartBodyBuilder which ensures an HttpEntity wrapper.

As an alternative to MultipartBodyBuilder, you can also provide multipart content, inline-style, through the built-in BodyInserters. For example:

import static org.springframework.web.reactive.function.BodyInserters.\*;

Mono<Void> result = client.post()

.uri("/path", id)

.body(fromMultipartData("fieldPart", "value").with("filePart", resource))

.retrieve()

.bodyToMono(Void.class);

2.4. Builder options

A simple way to create WebClient is through the static factory methods create() and create(String) with a base URL for all requests. You can also use WebClient.builder() for access to more options.

To customize the underlying HTTP client:

SslContext sslContext = ...

ClientHttpConnector connector = new ReactorClientHttpConnector(

builder -> builder.sslContext(sslContext));

WebClient webClient = WebClient.builder()

.clientConnector(connector)

.build();

To customize the HTTP codecs used for encoding and decoding HTTP messages:

ExchangeStrategies strategies = ExchangeStrategies.builder()

.codecs(configurer -> {

// ...

})

.build();

WebClient webClient = WebClient.builder()

.exchangeStrategies(strategies)

.build();

The builder can be used to insert Client Filters.

Explore the WebClient.Builder in your IDE for other options related to URI building, default headers (and cookies), and more.

After the WebClient is built, you can always obtain a new builder from it, in order to build a new WebClient, based on, but without affecting the current instance:

WebClient modifiedClient = client.mutate()

// user builder methods...

.build();

2.5. Client Filters

You can register an ExchangeFilterFunction in the WebClient.Builder to intercept and possibly modify requests performed through the client:

WebClient client = WebClient.builder()

.filter((request, next) -> {

ClientRequest filtered = ClientRequest.from(request)

.header("foo", "bar")

.build();

return next.exchange(filtered);

})

.build();

This can be used for cross-cutting concerns such as authentication. The example below uses a filter for basic authentication through a static factory method:

// static import of ExchangeFilterFunctions.basicAuthentication

WebClient client = WebClient.builder()

.filter(basicAuthentication("user", "password"))

.build();

Filters apply globally to every request. To change how a filter’s behavior for a specific request, you can add request attributes to the ClientRequest that can then be accessed by all filters in the chain:

WebClient client = WebClient.builder()

.filter((request, next) -> {

Optional<Object> usr = request.attribute("myAttribute");

// ...

})

.build();

client.get().uri("http://example.org/")

.attribute("myAttribute", "...")

.retrieve()

.bodyToMono(Void.class);

}

You can also replicate an existing WebClient, and insert new filters or remove already registered filters. In the example below, a basic authentication filter is inserted at index 0:

// static import of ExchangeFilterFunctions.basicAuthentication

WebClient client = webClient.mutate()

.filters(filterList -> {

filterList.add(0, basicAuthentication("user", "password"));

})

.build();

2.6. Testing

To test code that uses the WebClient, you can use a mock web server such as the OkHttp MockWebServer. To see example use, check WebClientIntegrationTests in the Spring Framework tests, or the static-server sample in the OkHttp repository.

3. WebSockets

Same in Servlet stack

This part of the reference documentation covers support for Reactive stack, WebSocket messaging.

3.1. Introduction

The WebSocket protocol RFC 6455 provides a standardized way to establish a full-duplex, two-way communication channel between client and server over a single TCP connection. It is a different TCP protocol from HTTP but is designed to work over HTTP, using ports 80 and 443 and allowing re-use of existing firewall rules.

A WebSocket interaction begins with an HTTP request that uses the HTTP "Upgrade" header to upgrade, or in this case to switch, to the WebSocket protocol:

GET /spring-websocket-portfolio/portfolio HTTP/1.1

Host: localhost:8080

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: Uc9l9TMkWGbHFD2qnFHltg==

Sec-WebSocket-Protocol: v10.stomp, v11.stomp

Sec-WebSocket-Version: 13

Origin: http://localhost:8080

Instead of the usual 200 status code, a server with WebSocket support returns:

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: 1qVdfYHU9hPOl4JYYNXF623Gzn0=

Sec-WebSocket-Protocol: v10.stomp

After a successful handshake the TCP socket underlying the HTTP upgrade request remains open for both client and server to continue to send and receive messages.

A complete introduction of how WebSockets work is beyond the scope of this document. Please read RFC 6455, the WebSocket chapter of HTML5, or one of many introductions and tutorials on the Web.

Note that if a WebSocket server is running behind a web server (e.g. nginx) you will likely need to configure it to pass WebSocket upgrade requests on to the WebSocket server. Likewise if the application runs in a cloud environment, check the instructions of the cloud provider related to WebSocket support.

3.1.1. HTTP vs WebSocket

Even though WebSocket is designed to be HTTP compatible and starts with an HTTP request, it is important to understand that the two protocols lead to very different architectures and application programming models.

In HTTP and REST, an application is modeled as many URLs. To interact with the application clients access those URLs, request-response style. Servers route requests to the appropriate handler based on the HTTP URL, method, and headers.

By contrast in WebSockets there is usually just one URL for the initial connect and subsequently all application messages flow on that same TCP connection. This points to an entirely different asynchronous, event-driven, messaging architecture.

WebSocket is also a low-level transport protocol which unlike HTTP does not prescribe any semantics to the content of messages. That means there is no way to route or process a message unless client and server agree on message semantics.

WebSocket clients and servers can negotiate the use of a higher-level, messaging protocol (e.g. STOMP), via the "Sec-WebSocket-Protocol" header on the HTTP handshake request, or in the absence of that they need to come up with their own conventions.

3.1.2. When to use it?

WebSockets can make a web page dynamic and interactive. However in many cases a combination of Ajax and HTTP streaming and/or long polling could provide a simple and effective solution.

For example news, mail, and social feeds need to update dynamically but it may be perfectly okay to do so every few minutes. Collaboration, games, and financial apps on the other hand need to be much closer to real time.

Latency alone is not a deciding factor. If the volume of messages is relatively low (e.g. monitoring network failures) HTTP streaming or polling may provide an effective solution. It is the combination of low latency, high frequency and high volume that make the best case for the use WebSocket.

Keep in mind also that over the Internet, restrictive proxies outside your control, may preclude WebSocket interactions either because they are not configured to pass on the Upgrade header or because they close long lived connections that appear idle? This means that the use of WebSocket for internal applications within the firewall is a more straight-forward decision than it is for public facing applications.

3.2. WebSocket API

Same in Servlet stack

The Spring Framework provides a WebSocket API that can be used to write client and server side applications that handle WebSocket messages.

3.2.1. Server

Same in Servlet stack

To create a WebSocket server, first create a WebSocketHandler:

import org.springframework.web.reactive.socket.WebSocketHandler;

import org.springframework.web.reactive.socket.WebSocketSession;

public class MyWebSocketHandler implements WebSocketHandler {

@Override

public Mono<Void> handle(WebSocketSession session) {

// ...

}

}

Then map it to a URL and add a WebSocketHandlerAdapter:

@Configuration

static class WebConfig {

@Bean

public HandlerMapping handlerMapping() {

Map<String, WebSocketHandler> map = new HashMap<>();

map.put("/path", new MyWebSocketHandler());

SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();

mapping.setUrlMap(map);

mapping.setOrder(-1); // before annotated controllers

return mapping;

}

@Bean

public WebSocketHandlerAdapter handlerAdapter() {

return new WebSocketHandlerAdapter();

}

}

3.2.2. WebSocketHandler

The handle method of WebSocketHandler takes WebSocketSession and returns Mono<Void> to indicate when application handling of the session is complete. The session is handled through two streams, one for inbound and one for outbound messages:

WebSocketSession method Description

Flux<WebSocketMessage> receive()

Provides access to the inbound message stream, and completes when the connection is closed.

Mono<Void> send(Publisher<WebSocketMessage>)

Takes a source for outgoing messages, writes the messages, and returns a Mono<Void> that completes when the source completes and writing is done.

A WebSocketHandler must compose the inbound and outbound streams into a unified flow, and return a Mono<Void> that reflects the completion of that flow. Depending on application requirements, the unified flow completes when:

Either inbound or outbound message streams complete.

Inbound stream completes (i.e. connection closed), while outbound is infinite.

At a chosen point through the close method of WebSocketSession.

When inbound and outbound message streams are composed together, there is no need to check if the connection is open, since Reactive Streams signals will terminate activity. The inbound stream receives a completion/error signal, and the outbound stream receives receives a cancellation signal.

The most basic implementation of a handler is one that handles the inbound stream:

class ExampleHandler implements WebSocketHandler {

@Override

public Mono<Void> handle(WebSocketSession session) {

return session.receive()

.doOnNext(message -> {

// ...

})

.concatMap(message -> {

// ...

})

.then();

}

}

Access stream of inbound messages.

Do something with each message.

Perform nested async operation using message content.

Return Mono<Void> that completes when receiving completes.

For nested, asynchronous operations, you may need to call message.retain() on underlying servers that use pooled data buffers (e.g. Netty), or otherwise the data buffer may be released before you’ve had a chance to read the data. For more background see Data Buffers and Codecs.

The below implementation combines the inbound with the outbound streams:

class ExampleHandler implements WebSocketHandler {

@Override

public Mono<Void> handle(WebSocketSession session) {

Flux<WebSocketMessage> output = session.receive()

.doOnNext(message -> {

// ...

})

.concatMap(message -> {

// ...

})

.map(value -> session.textMessage("Echo " + value));

return session.send(output);

}

}

Handle inbound message stream.

Create outbound message, producing a combined flow.

Return Mono<Void> that doesn’t complete while we continue to receive.

Inbound and outbound streams can be independent, and joined only for completion:

class ExampleHandler implements WebSocketHandler {

@Override

public Mono<Void> handle(WebSocketSession session) {

Mono<Void> input = session.receive()

.doOnNext(message -> {

// ...

})

.concatMap(message -> {

// ...

})

.then();

Flux<String> source = ... ;

Mono<Void> output = session.send(source.map(session::textMessage));

return Mono.zip(input, output).then();

}

}

Handle inbound message stream.

Send outgoing messages.

Join the streams and return Mono<Void> that completes when either stream ends.

3.2.3. Handshake

Same in Servlet stack

WebSocketHandlerAdapter delegates to a WebSocketService. By default that’s an instance of HandshakeWebSocketService, which performs basic checks on the WebSocket request and then uses RequestUpgradeStrategy for the server in use. Currently there is built-in support for Reactor Netty, Tomcat, Jetty, and Undertow.

The above are just 3 examples to serve as a starting point.

3.2.4. Server config

Same in Servlet stack

The RequestUpgradeStrategy for each server exposes the WebSocket-related configuration options available for the underlying WebSocket engine. Below is an example of setting WebSocket options when running on Tomcat:

@Configuration

static class WebConfig {

@Bean

public WebSocketHandlerAdapter handlerAdapter() {

return new WebSocketHandlerAdapter(webSocketService());

}

@Bean

public WebSocketService webSocketService() {

TomcatRequestUpgradeStrategy strategy = new TomcatRequestUpgradeStrategy();

strategy.setMaxSessionIdleTimeout(0L);

return new HandshakeWebSocketService(strategy);

}

}

Check the upgrade strategy for your server to see what options are available. Currently only Tomcat and Jetty expose such options.

3.2.5. CORS

Same in Servlet stack

The easiest way to configure CORS and restrict access to a WebSocket endpoint is to have your WebSocketHandler implement CorsConfigurationSource and return a CorsConfiguraiton with allowed origins, headers, etc. If for any reason you can’t do that, you can also set the corsConfigurations property on the SimpleUrlHandler to specify CORS settings by URL pattern. If both are specified they’re combined via the combine method on CorsConfiguration.

3.2.6. Client

Spring WebFlux provides a WebSocketClient abstraction with implementations for Reactor Netty, Tomcat, Jetty, Undertow, and standard Java (i.e. JSR-356).

The Tomcat client is effectively an extension of the standard Java one with some extra functionality in the WebSocketSession handling taking advantage of Tomcat specific API to suspend receiving messages for back pressure.

To start a WebSocket session, create an instance of the client and use its execute methods:

WebSocketClient client = new ReactorNettyWebSocketClient();

URI url = new URI("ws://localhost:8080/path");

client.execute(url, session ->

session.receive()

.doOnNext(System.out::println)

.then());

Some clients, e.g. Jetty, implement Lifecycle and need to be started in stopped before you can use them. All clients have constructor options related to configuration of the underlying WebSocket client.

4. Testing

Same in Spring MVC

The spring-test module provides mock implementations of ServerHttpRequest, ServerHttpResponse, and ServerWebExchange. See Spring Web Reactive mock objects.

The WebTestClient builds on these mock request and response objects to provide support for testing WebFlux applications without and HTTP server. The WebTestClient can be used for end-to-end integration tests too.

4.1. Threading model

5. Reactive Libraries

spring-webflux depends on reactor-core and uses it internally to compose asynchronous logic and to provide Reactive Streams support. Generally WebFlux APIs return Flux or Mono — since that’s what’s used internally, and leniently accept any Reactive Streams Publisher implementation as input. The use of Flux vs Mono is important because it helps to express cardinality — e.g. whether a single or multiple async values are expected, and that can be essential for making decisions, for example when encoding or decoding HTTP messages.

For annotated controllers, WebFlux transparently adapts to the reactive library chosen by the application. This is done with the help of the ReactiveAdapterRegistry which provides pluggable support for reactive library and other asynchronous types. The registry has built-in support for RxJava and CompletableFuture, but others can be registered too.

For functional APIs such as Functional Endpoints, the WebClient, and others, the general rules for WebFlux APIs apply — Flux and Mono as return values, and Reactive Streams Publisher as input. When a Publisher, whether custom or from another reactive library, is provided, it can only be treated as a stream with unknown semantics (0..N). If however the semantics are known, you can wrap it with Flux or Mono.from(Publisher) instead of passing the raw Publisher.

For example, given a Publisher that is not a Mono, the Jackson JSON message writer expects multiple values. If the media type implies an infinite stream — e.g. "application/json+stream", values are written and flushed individually; otherwise values are buffered into a list and rendered as a JSON array.

## Integration

Version 5.0.8.RELEASE

This part of the reference documentation covers the Spring Framework’s integration with a number of Java EE (and related) technologies.

1. Remoting and web services using Spring

1.1. Introduction

Spring features integration classes for remoting support using various technologies. The remoting support eases the development of remote-enabled services, implemented by your usual (Spring) POJOs. Currently, Spring supports the following remoting technologies:

Remote Method Invocation (RMI). Through the use of the RmiProxyFactoryBean and the RmiServiceExporter Spring supports both traditional RMI (with java.rmi.Remote interfaces and java.rmi.RemoteException) and transparent remoting via RMI invokers (with any Java interface).

Spring’s HTTP invoker. Spring provides a special remoting strategy which allows for Java serialization via HTTP, supporting any Java interface (just like the RMI invoker). The corresponding support classes are HttpInvokerProxyFactoryBean and HttpInvokerServiceExporter.

Hessian. By using Spring’s HessianProxyFactoryBean and the HessianServiceExporter you can transparently expose your services using the lightweight binary HTTP-based protocol provided by Caucho.

JAX-WS. Spring provides remoting support for web services via JAX-WS (the successor of JAX-RPC, as introduced in Java EE 5 and Java 6).

JMS. Remoting using JMS as the underlying protocol is supported via the JmsInvokerServiceExporter and JmsInvokerProxyFactoryBean classes.

AMQP. Remoting using AMQP as the underlying protocol is supported by the Spring AMQP project.

While discussing the remoting capabilities of Spring, we’ll use the following domain model and corresponding services:

public class Account implements Serializable{

private String name;

public String getName(){

return name;

}

public void setName(String name) {

this.name = name;

}

}

public interface AccountService {

public void insertAccount(Account account);

public List<Account> getAccounts(String name);

}

// the implementation doing nothing at the moment

public class AccountServiceImpl implements AccountService {

public void insertAccount(Account acc) {

// do something...

}

public List<Account> getAccounts(String name) {

// do something...

}

}

We will start exposing the service to a remote client by using RMI and talk a bit about the drawbacks of using RMI. We’ll then continue to show an example using Hessian as the protocol.

1.2. Exposing services using RMI

Using Spring’s support for RMI, you can transparently expose your services through the RMI infrastructure. After having this set up, you basically have a configuration similar to remote EJBs, except for the fact that there is no standard support for security context propagation or remote transaction propagation. Spring does provide hooks for such additional invocation context when using the RMI invoker, so you can for example plug in security frameworks or custom security credentials here.

1.2.1. Exporting the service using the RmiServiceExporter

Using the RmiServiceExporter, we can expose the interface of our AccountService object as RMI object. The interface can be accessed by using RmiProxyFactoryBean, or via plain RMI in case of a traditional RMI service. The RmiServiceExporter explicitly supports the exposing of any non-RMI services via RMI invokers.

Of course, we first have to set up our service in the Spring container:

<bean id="accountService" class="example.AccountServiceImpl">

<!-- any additional properties, maybe a DAO? -->

</bean>

Next we’ll have to expose our service using the RmiServiceExporter:

<bean class="org.springframework.remoting.rmi.RmiServiceExporter">

<!-- does not necessarily have to be the same name as the bean to be exported -->

<property name="serviceName" value="AccountService"/>

<property name="service" ref="accountService"/>

<property name="serviceInterface" value="example.AccountService"/>

<!-- defaults to 1099 -->

<property name="registryPort" value="1199"/>

</bean>

As you can see, we’re overriding the port for the RMI registry. Often, your application server also maintains an RMI registry and it is wise to not interfere with that one. Furthermore, the service name is used to bind the service under. So right now, the service will be bound at 'rmi://HOST:1199/AccountService'. We’ll use the URL later on to link in the service at the client side.

The servicePort property has been omitted (it defaults to 0). This means that an anonymous port will be used to communicate with the service.

1.2.2. Linking in the service at the client

Our client is a simple object using the AccountService to manage accounts:

public class SimpleObject {

private AccountService accountService;

public void setAccountService(AccountService accountService) {

this.accountService = accountService;

}

// additional methods using the accountService

}

To link in the service on the client, we’ll create a separate Spring container, containing the simple object and the service linking configuration bits:

<bean class="example.SimpleObject">

<property name="accountService" ref="accountService"/>

</bean>

<bean id="accountService" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">

<property name="serviceUrl" value="rmi://HOST:1199/AccountService"/>

<property name="serviceInterface" value="example.AccountService"/>

</bean>

That’s all we need to do to support the remote account service on the client. Spring will transparently create an invoker and remotely enable the account service through the RmiServiceExporter. At the client we’re linking it in using the RmiProxyFactoryBean.

1.3. Using Hessian to remotely call services via HTTP

Hessian offers a binary HTTP-based remoting protocol. It is developed by Caucho and more information about Hessian itself can be found at http://www.caucho.com.

1.3.1. Wiring up the DispatcherServlet for Hessian and co.

Hessian communicates via HTTP and does so using a custom servlet. Using Spring’s DispatcherServlet principles, as known from Spring Web MVC usage, you can easily wire up such a servlet exposing your services. First we’ll have to create a new servlet in your application (this is an excerpt from 'web.xml'):

<servlet>

<servlet-name>remoting</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>remoting</servlet-name>

<url-pattern>/remoting/\*</url-pattern>

</servlet-mapping>

You’re probably familiar with Spring’s DispatcherServlet principles and if so, you know that now you’ll have to create a Spring container configuration resource named 'remoting-servlet.xml' (after the name of your servlet) in the 'WEB-INF' directory. The application context will be used in the next section.

Alternatively, consider the use of Spring’s simpler HttpRequestHandlerServlet. This allows you to embed the remote exporter definitions in your root application context (by default in 'WEB-INF/applicationContext.xml'), with individual servlet definitions pointing to specific exporter beans. Each servlet name needs to match the bean name of its target exporter in this case.

1.3.2. Exposing your beans by using the HessianServiceExporter

In the newly created application context called remoting-servlet.xml, we’ll create a HessianServiceExporter exporting your services:

<bean id="accountService" class="example.AccountServiceImpl">

<!-- any additional properties, maybe a DAO? -->

</bean>

<bean name="/AccountService" class="org.springframework.remoting.caucho.HessianServiceExporter">

<property name="service" ref="accountService"/>

<property name="serviceInterface" value="example.AccountService"/>

</bean>

Now we’re ready to link in the service at the client. No explicit handler mapping is specified, mapping request URLs onto services, so BeanNameUrlHandlerMapping will be used: Hence, the service will be exported at the URL indicated through its bean name within the containing DispatcherServlet's mapping (as defined above): 'http://HOST:8080/remoting/AccountService'.

Alternatively, create a HessianServiceExporter in your root application context (e.g. in 'WEB-INF/applicationContext.xml'):

<bean name="accountExporter" class="org.springframework.remoting.caucho.HessianServiceExporter">

<property name="service" ref="accountService"/>

<property name="serviceInterface" value="example.AccountService"/>

</bean>

In the latter case, define a corresponding servlet for this exporter in 'web.xml', with the same end result: The exporter getting mapped to the request path /remoting/AccountService. Note that the servlet name needs to match the bean name of the target exporter.

<servlet>

<servlet-name>accountExporter</servlet-name>

<servlet-class>org.springframework.web.context.support.HttpRequestHandlerServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>accountExporter</servlet-name>

<url-pattern>/remoting/AccountService</url-pattern>

</servlet-mapping>

1.3.3. Linking in the service on the client

Using the HessianProxyFactoryBean we can link in the service at the client. The same principles apply as with the RMI example. We’ll create a separate bean factory or application context and mention the following beans where the SimpleObject is using the AccountService to manage accounts:

<bean class="example.SimpleObject">

<property name="accountService" ref="accountService"/>

</bean>

<bean id="accountService" class="org.springframework.remoting.caucho.HessianProxyFactoryBean">

<property name="serviceUrl" value="http://remotehost:8080/remoting/AccountService"/>

<property name="serviceInterface" value="example.AccountService"/>

</bean>

1.3.4. Applying HTTP basic authentication to a service exposed through Hessian

One of the advantages of Hessian is that we can easily apply HTTP basic authentication, because both protocols are HTTP-based. Your normal HTTP server security mechanism can easily be applied through using the web.xml security features, for example. Usually, you don’t use per-user security credentials here, but rather shared credentials defined at the HessianProxyFactoryBean level (similar to a JDBC DataSource).

<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">

<property name="interceptors" ref="authorizationInterceptor"/>

</bean>

<bean id="authorizationInterceptor"

class="org.springframework.web.servlet.handler.UserRoleAuthorizationInterceptor">

<property name="authorizedRoles" value="administrator,operator"/>

</bean>

This is an example where we explicitly mention the BeanNameUrlHandlerMapping and set an interceptor allowing only administrators and operators to call the beans mentioned in this application context.

Of course, this example doesn’t show a flexible kind of security infrastructure. For more options as far as security is concerned, have a look at the Spring Security project at http://projects.spring.io/spring-security/.

1.4. Exposing services using HTTP invokers

As opposed to Hessian, which are both lightweight protocols using their own slim serialization mechanisms, Spring HTTP invokers use the standard Java serialization mechanism to expose services through HTTP. This has a huge advantage if your arguments and return types are complex types that cannot be serialized using the serialization mechanisms Hessian uses (refer to the next section for more considerations when choosing a remoting technology).

Under the hood, Spring uses either the standard facilities provided by the JDK or Apache HttpComponents to perform HTTP calls. Use the latter if you need more advanced and easier-to-use functionality. Refer to hc.apache.org/httpcomponents-client-ga/ for more information.

Be aware of vulnerabilities due to unsafe Java deserialization: Manipulated input streams could lead to unwanted code execution on the server during the deserialization step. As a consequence, do not expose HTTP invoker endpoints to untrusted clients but rather just between your own services. In general, we strongly recommend any other message format (e.g. JSON) instead.

If you are concerned about security vulnerabilities due to Java serialization, consider the general-purpose serialization filter mechanism at the core JVM level, originally developed for JDK 9 but backported to JDK 8, 7 and 6 in the meantime: https://blogs.oracle.com/java-platform-group/entry/incoming\_filter\_serialization\_data\_a http://openjdk.java.net/jeps/290

1.4.1. Exposing the service object

Setting up the HTTP invoker infrastructure for a service object resembles closely the way you would do the same using Hessian. Just as Hessian support provides the HessianServiceExporter, Spring’s HttpInvoker support provides the org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter.

To expose the AccountService (mentioned above) within a Spring Web MVC DispatcherServlet, the following configuration needs to be in place in the dispatcher’s application context:

<bean name="/AccountService" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">

<property name="service" ref="accountService"/>

<property name="serviceInterface" value="example.AccountService"/>

</bean>

Such an exporter definition will be exposed through the DispatcherServlet's standard mapping facilities, as explained in the section on Hessian.

Alternatively, create an HttpInvokerServiceExporter in your root application context (e.g. in 'WEB-INF/applicationContext.xml'):

<bean name="accountExporter" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">

<property name="service" ref="accountService"/>

<property name="serviceInterface" value="example.AccountService"/>

</bean>

In addition, define a corresponding servlet for this exporter in 'web.xml', with the servlet name matching the bean name of the target exporter:

<servlet>

<servlet-name>accountExporter</servlet-name>

<servlet-class>org.springframework.web.context.support.HttpRequestHandlerServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>accountExporter</servlet-name>

<url-pattern>/remoting/AccountService</url-pattern>

</servlet-mapping>

If you are running outside of a servlet container and are using Oracle’s Java 6, then you can use the built-in HTTP server implementation. You can configure the SimpleHttpServerFactoryBean together with a SimpleHttpInvokerServiceExporter as is shown in this example:

<bean name="accountExporter"

class="org.springframework.remoting.httpinvoker.SimpleHttpInvokerServiceExporter">

<property name="service" ref="accountService"/>

<property name="serviceInterface" value="example.AccountService"/>

</bean>

<bean id="httpServer"

class="org.springframework.remoting.support.SimpleHttpServerFactoryBean">

<property name="contexts">

<util:map>

<entry key="/remoting/AccountService" value-ref="accountExporter"/>

</util:map>

</property>

<property name="port" value="8080"/>

</bean>

1.4.2. Linking in the service at the client

Again, linking in the service from the client much resembles the way you would do it when using Hessian. Using a proxy, Spring will be able to translate your calls to HTTP POST requests to the URL pointing to the exported service.

<bean id="httpInvokerProxy" class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">

<property name="serviceUrl" value="http://remotehost:8080/remoting/AccountService"/>

<property name="serviceInterface" value="example.AccountService"/>

</bean>

As mentioned before, you can choose what HTTP client you want to use. By default, the HttpInvokerProxy uses the JDK’s HTTP functionality, but you can also use the Apache HttpComponents client by setting the httpInvokerRequestExecutor property:

<property name="httpInvokerRequestExecutor">

<bean class="org.springframework.remoting.httpinvoker.HttpComponentsHttpInvokerRequestExecutor"/>

</property>

1.5. Web services

Spring provides full support for standard Java web services APIs:

Exposing web services using JAX-WS

Accessing web services using JAX-WS

In addition to stock support for JAX-WS in Spring Core, the Spring portfolio also features Spring Web Services, a solution for contract-first, document-driven web services - highly recommended for building modern, future-proof web services.

1.5.1. Exposing servlet-based web services using JAX-WS

Spring provides a convenient base class for JAX-WS servlet endpoint implementations - SpringBeanAutowiringSupport. To expose our AccountService we extend Spring’s SpringBeanAutowiringSupport class and implement our business logic here, usually delegating the call to the business layer. We’ll simply use Spring’s @Autowired annotation for expressing such dependencies on Spring-managed beans.

/\*\*

\* JAX-WS compliant AccountService implementation that simply delegates

\* to the AccountService implementation in the root web application context.

\*

\* This wrapper class is necessary because JAX-WS requires working with dedicated

\* endpoint classes. If an existing service needs to be exported, a wrapper that

\* extends SpringBeanAutowiringSupport for simple Spring bean autowiring (through

\* the @Autowired annotation) is the simplest JAX-WS compliant way.

\*

\* This is the class registered with the server-side JAX-WS implementation.

\* In the case of a Java EE 5 server, this would simply be defined as a servlet

\* in web.xml, with the server detecting that this is a JAX-WS endpoint and reacting

\* accordingly. The servlet name usually needs to match the specified WS service name.

\*

\* The web service engine manages the lifecycle of instances of this class.

\* Spring bean references will just be wired in here.

\*/

import org.springframework.web.context.support.SpringBeanAutowiringSupport;

@WebService(serviceName="AccountService")

public class AccountServiceEndpoint extends SpringBeanAutowiringSupport {

@Autowired

private AccountService biz;

@WebMethod

public void insertAccount(Account acc) {

biz.insertAccount(acc);

}

@WebMethod

public Account[] getAccounts(String name) {

return biz.getAccounts(name);

}

}

Our AccountServiceEndpoint needs to run in the same web application as the Spring context to allow for access to Spring’s facilities. This is the case by default in Java EE 5 environments, using the standard contract for JAX-WS servlet endpoint deployment. See Java EE 5 web service tutorials for details.

1.5.2. Exporting standalone web services using JAX-WS

The built-in JAX-WS provider that comes with Oracle’s JDK supports exposure of web services using the built-in HTTP server that’s included in the JDK as well. Spring’s SimpleJaxWsServiceExporter detects all @WebService annotated beans in the Spring application context, exporting them through the default JAX-WS server (the JDK HTTP server).

In this scenario, the endpoint instances are defined and managed as Spring beans themselves; they will be registered with the JAX-WS engine but their lifecycle will be up to the Spring application context. This means that Spring functionality like explicit dependency injection may be applied to the endpoint instances. Of course, annotation-driven injection through @Autowired will work as well.

<bean class="org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter">

<property name="baseAddress" value="http://localhost:8080/"/>

</bean>

<bean id="accountServiceEndpoint" class="example.AccountServiceEndpoint">

...

</bean>

...

The AccountServiceEndpoint may derive from Spring’s SpringBeanAutowiringSupport but doesn’t have to since the endpoint is a fully Spring-managed bean here. This means that the endpoint implementation may look like as follows, without any superclass declared - and Spring’s @Autowired configuration annotation still being honored:

@WebService(serviceName="AccountService")

public class AccountServiceEndpoint {

@Autowired

private AccountService biz;

@WebMethod

public void insertAccount(Account acc) {

biz.insertAccount(acc);

}

@WebMethod

public List<Account> getAccounts(String name) {

return biz.getAccounts(name);

}

}

1.5.3. Exporting web services using the JAX-WS RI’s Spring support

Oracle’s JAX-WS RI, developed as part of the GlassFish project, ships Spring support as part of its JAX-WS Commons project. This allows for defining JAX-WS endpoints as Spring-managed beans, similar to the standalone mode discussed in the previous section - but this time in a Servlet environment. Note that this is not portable in a Java EE 5 environment; it is mainly intended for non-EE environments such as Tomcat, embedding the JAX-WS RI as part of the web application.

The difference to the standard style of exporting servlet-based endpoints is that the lifecycle of the endpoint instances themselves will be managed by Spring here, and that there will be only one JAX-WS servlet defined in web.xml. With the standard Java EE 5 style (as illustrated above), you’ll have one servlet definition per service endpoint, with each endpoint typically delegating to Spring beans (through the use of @Autowired, as shown above).

Check out https://jax-ws-commons.java.net/spring/ for details on setup and usage style.

1.5.4. Accessing web services using JAX-WS

Spring provides two factory beans to create JAX-WS web service proxies, namely LocalJaxWsServiceFactoryBean and JaxWsPortProxyFactoryBean. The former can only return a JAX-WS service class for us to work with. The latter is the full-fledged version that can return a proxy that implements our business service interface. In this example we use the latter to create a proxy for the AccountService endpoint (again):

<bean id="accountWebService" class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean">

<property name="serviceInterface" value="example.AccountService"/>

<property name="wsdlDocumentUrl" value="http://localhost:8888/AccountServiceEndpoint?WSDL"/>

<property name="namespaceUri" value="http://example/"/>

<property name="serviceName" value="AccountService"/>

<property name="portName" value="AccountServiceEndpointPort"/>

</bean>

Where serviceInterface is our business interface the clients will use. wsdlDocumentUrl is the URL for the WSDL file. Spring needs this a startup time to create the JAX-WS Service. namespaceUri corresponds to the targetNamespace in the .wsdl file. serviceName corresponds to the service name in the .wsdl file. portName corresponds to the port name in the .wsdl file.

Accessing the web service is now very easy as we have a bean factory for it that will expose it as AccountService interface. We can wire this up in Spring:

<bean id="client" class="example.AccountClientImpl">

...

<property name="service" ref="accountWebService"/>

</bean>

From the client code we can access the web service just as if it was a normal class:

public class AccountClientImpl {

private AccountService service;

public void setService(AccountService service) {

this.service = service;

}

public void foo() {

service.insertAccount(...);

}

}

The above is slightly simplified in that JAX-WS requires endpoint interfaces and implementation classes to be annotated with @WebService, @SOAPBinding etc annotations. This means that you cannot (easily) use plain Java interfaces and implementation classes as JAX-WS endpoint artifacts; you need to annotate them accordingly first. Check the JAX-WS documentation for details on those requirements.

1.6. JMS

It is also possible to expose services transparently using JMS as the underlying communication protocol. The JMS remoting support in the Spring Framework is pretty basic - it sends and receives on the same thread and in the same non-transactional Session, and as such throughput will be very implementation dependent. Note that these single-threaded and non-transactional constraints apply only to Spring’s JMS remoting support. See JMS (Java Message Service) for information on Spring’s rich support for JMS-based messaging.

The following interface is used on both the server and the client side.

package com.foo;

public interface CheckingAccountService {

public void cancelAccount(Long accountId);

}

The following simple implementation of the above interface is used on the server-side.

package com.foo;

public class SimpleCheckingAccountService implements CheckingAccountService {

public void cancelAccount(Long accountId) {

System.out.println("Cancelling account [" + accountId + "]");

}

}

This configuration file contains the JMS-infrastructure beans that are shared on both the client and server.

<?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="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">

<property name="brokerURL" value="tcp://ep-t43:61616"/>

</bean>

<bean id="queue" class="org.apache.activemq.command.ActiveMQQueue">

<constructor-arg value="mmm"/>

</bean>

</beans>

1.6.1. Server-side configuration

On the server, you just need to expose the service object using the JmsInvokerServiceExporter.

<?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="checkingAccountService"

class="org.springframework.jms.remoting.JmsInvokerServiceExporter">

<property name="serviceInterface" value="com.foo.CheckingAccountService"/>

<property name="service">

<bean class="com.foo.SimpleCheckingAccountService"/>

</property>

</bean>

<bean class="org.springframework.jms.listener.SimpleMessageListenerContainer">

<property name="connectionFactory" ref="connectionFactory"/>

<property name="destination" ref="queue"/>

<property name="concurrentConsumers" value="3"/>

<property name="messageListener" ref="checkingAccountService"/>

</bean>

</beans>

package com.foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Server {

public static void main(String[] args) throws Exception {

new ClassPathXmlApplicationContext(new String[]{"com/foo/server.xml", "com/foo/jms.xml"});

}

}

1.6.2. Client-side configuration

The client merely needs to create a client-side proxy that will implement the agreed upon interface ( CheckingAccountService). The resulting object created off the back of the following bean definition can be injected into other client side objects, and the proxy will take care of forwarding the call to the server-side object via JMS.

<?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="checkingAccountService"

class="org.springframework.jms.remoting.JmsInvokerProxyFactoryBean">

<property name="serviceInterface" value="com.foo.CheckingAccountService"/>

<property name="connectionFactory" ref="connectionFactory"/>

<property name="queue" ref="queue"/>

</bean>

</beans>

package com.foo;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Client {

public static void main(String[] args) throws Exception {

ApplicationContext ctx = new ClassPathXmlApplicationContext(

new String[] {"com/foo/client.xml", "com/foo/jms.xml"});

CheckingAccountService service = (CheckingAccountService) ctx.getBean("checkingAccountService");

service.cancelAccount(new Long(10));

}

}

1.7. AMQP

Refer to the Spring AMQP Reference Document 'Spring Remoting with AMQP' section for more information.

1.8. Auto-detection is not implemented for remote interfaces

The main reason why auto-detection of implemented interfaces does not occur for remote interfaces is to avoid opening too many doors to remote callers. The target object might implement internal callback interfaces like InitializingBean or DisposableBean which one would not want to expose to callers.

Offering a proxy with all interfaces implemented by the target usually does not matter in the local case. But when exporting a remote service, you should expose a specific service interface, with specific operations intended for remote usage. Besides internal callback interfaces, the target might implement multiple business interfaces, with just one of them intended for remote exposure. For these reasons, we require such a service interface to be specified.

This is a trade-off between configuration convenience and the risk of accidental exposure of internal methods. Always specifying a service interface is not too much effort, and puts you on the safe side regarding controlled exposure of specific methods.

1.9. Considerations when choosing a technology

Each and every technology presented here has its drawbacks. You should carefully consider your needs, the services you are exposing and the objects you’ll be sending over the wire when choosing a technology.

When using RMI, it’s not possible to access the objects through the HTTP protocol, unless you’re tunneling the RMI traffic. RMI is a fairly heavy-weight protocol in that it supports full-object serialization which is important when using a complex data model that needs serialization over the wire. However, RMI-JRMP is tied to Java clients: It is a Java-to-Java remoting solution.

Spring’s HTTP invoker is a good choice if you need HTTP-based remoting but also rely on Java serialization. It shares the basic infrastructure with RMI invokers, just using HTTP as transport. Note that HTTP invokers are not only limited to Java-to-Java remoting but also to Spring on both the client and server side. (The latter also applies to Spring’s RMI invoker for non-RMI interfaces.)

Hessian might provide significant value when operating in a heterogeneous environment, because they explicitly allow for non-Java clients. However, non-Java support is still limited. Known issues include the serialization of Hibernate objects in combination with lazily-initialized collections. If you have such a data model, consider using RMI or HTTP invokers instead of Hessian.

JMS can be useful for providing clusters of services and allowing the JMS broker to take care of load balancing, discovery and auto-failover. By default: Java serialization is used when using JMS remoting but the JMS provider could use a different mechanism for the wire formatting, such as XStream to allow servers to be implemented in other technologies.

Last but not least, EJB has an advantage over RMI in that it supports standard role-based authentication and authorization and remote transaction propagation. It is possible to get RMI invokers or HTTP invokers to support security context propagation as well, although this is not provided by core Spring: There are just appropriate hooks for plugging in third-party or custom solutions here.

1.10. REST Endpoints

The Spring Framework provides two choices for making calls to REST endpoints:

RestTemplate — the original Spring REST client with an API similar to other template classes in Spring such as JdbcTemplate, JmsTemplate and others. RestTemplate has a synchronous API and relies on blocking I/O which is okay for client scenarios with low concurrency.

WebClient — reactive client with a functional, fluent API from the spring-webflux module. It relies on non-blocking I/O which allows it to support high concurrency more efficiently (i.e. using a small number of threads) than the RestTemplate. WebClient is a natural fit for streaming scenarios.

1.10.1. RestTemplate

The RestTemplate provides a higher level API over HTTP client libraries. It makes it easy to invoke REST endpoints in a single line. It exposes the following groups of overloaded methods:

Table 1. RestTemplate methods

Method group Description

getForObject

Retrieve a representation via GET.

getForEntity

Retrieve a ResponseEntity, i.e. status, headers, and body, via GET.

headForHeaders

Retrieve all headers for a resource via HEAD.

postForLocation

Create a new resource via POST and return the Location header from the response.

postForObject

Create a new resource via POST and return the representation from the response.

postForEntity

Create a new resource via POST and return the representation from the response.

put

Create or update a resource via PUT.

patchForObject

Update a resource via PATCH and return the representation from the response. Note that the JDK HttpURLConnection does not support the PATCH but Apache HttpComponents, and others do.

delete

Delete the resources at the specified URI via DELETE.

optionsForAllow

Retrieve allowed HTTP methods for a resource via ALLOW.

exchange

More generalized, and less opinionated version, of the above methods that provides extra flexibility when needed. It accepts RequestEntity, including HTTP method, URL, headers, and body as input, and returns a ResponseEntity.

These methods allow the use of ParameterizedTypeReference instead of Class to specify a response type with generics.

execute

The most generalized way to perform a request, with full control over request preparation and response extraction via callback interfaces.

Initialization

The default constructor uses java.net.HttpURLConnection to perform requests. You can switch to a different HTTP library with an implementation of ClientHttpRequestFactory. There is built-in support for the following:

Apache HttpComponents

Netty

OkHttp

For example to switch to Apache HttpComponents use:

RestTemplate template = new RestTemplate(new HttpComponentsClientHttpRequestFactory());

Each ClientHttpRequestFactory exposes configuration options specific to the underlying HTTP client library, e.g. for credentials, connection pooling, etc.

Note that the java.net implementation for HTTP requests may raise an exception when accessing the status of a response that represents an error (e.g. 401). If this is an issue, switch to another HTTP client library.

URIs

Many of the RestTemplate methods accepts a URI template and URI template variables, either as a String vararg, or as Map<String,String>.

For example with a String vararg:

String result = restTemplate.getForObject(

"http://example.com/hotels/{hotel}/bookings/{booking}", String.class, "42", "21");

Or with a Map<String, String>:

Map<String, String> vars = Collections.singletonMap("hotel", "42");

String result = restTemplate.getForObject(

"http://example.com/hotels/{hotel}/rooms/{hotel}", String.class, vars);

Keep in mind URI templates are automatically encoded. For example:

restTemplate.getForObject("http://example.com/hotel list", String.class);

// Results in request to "http://example.com/hotel%20list"

You can use the uriTemplateHandler property of RestTemplate to customize how URIs are encoded. Or you can prepare a java.net.URI and pass it into one of the RestTemplate methods that accept a URI.

For more details on working with and encoding URIs, see URI Links.

Headers

Use the exchange() methods to specify request headers. For example:

String uriTemplate = "http://example.com/hotels/{hotel}";

URI uri = UriComponentsBuilder.fromUriString(uriTemplate).build(42);

RequestEntity<Void> requestEntity = RequestEntity.get(uri)

.header(("MyRequestHeader", "MyValue")

.build();

ResponseEntity<String> response = template.exchange(requestEntity, String.class);

String responseHeader = response.getHeaders().getFirst("MyResponseHeader");

String body = response.getBody();

Response headers can be obtained through many RestTemplate method variants that return ResponseEntity.

Body

Object passed into and returned from RestTemplate methods are converted to and from raw content with the help of an HttpMessageConverter.

On a POST, an input object is serialized to the request body:

URI location = template.postForLocation("http://example.com/people", person);

The "Content-Type" header of the request does not need to be set explicitly. In most cases a compatible message converter can be found based on the source Object type, and the chosen message converter will set the content type accordingly. If necessary, you can use the exchange methods to provide the "Content-Type" request header explicitly, and that in turn will influence what message converter is selected.

On a GET, the body of the response is deserialized to an output Object:

Person person = restTemplate.getForObject("http://example.com/people/{id}", Person.class, 42);

The "Accept" header of the request does not need to be set explicitly. In most cases a compatible message converter can be found based on the expected response type, which then helps to populate the "Accept" header. If necessary, you can use the exchange methods to provide the "Accept" header explicitly.

By default RestTemplate registers all built-in message converters, depending on classpath checks that help to determine what optional conversion libraries are present. You can also set the message converters to use explicitly.

Message Conversion

Same in Spring WebFlux

The spring-web module contains the HttpMessageConverter contract for reading and writing the body of HTTP requests and responses via InputStream and OutputStream. HttpMessageConverter's are used on the client side, e.g. in the RestTemplate, and also on the server side, e.g. in Spring MVC REST controllers.

Concrete implementations for the main media (MIME) types are provided in the framework and are registered by default with the RestTemplate on the client-side and with RequestMethodHandlerAdapter on the server-side (see Configuring Message Converters).

The implementations of HttpMessageConverters are described in the following sections. For all converters a default media type is used but can be overridden by setting the supportedMediaTypes bean property

Table 2. HttpMessageConverter Implementations

MessageConverter Description

StringHttpMessageConverter

An HttpMessageConverter implementation that can read and write Strings from the HTTP request and response. By default, this converter supports all text media types ( text/\*), and writes with a Content-Type of text/plain.

FormHttpMessageConverter

An HttpMessageConverter implementation that can read and write form data from the HTTP request and response. By default, this converter reads and writes the media type application/x-www-form-urlencoded. Form data is read from and written into a MultiValueMap<String, String>.

ByteArrayHttpMessageConverter

An HttpMessageConverter implementation that can read and write byte arrays from the HTTP request and response. By default, this converter supports all media types ( \*/\*), and writes with a Content-Type of application/octet-stream. This can be overridden by setting the supportedMediaTypes property, and overriding getContentType(byte[]).

MarshallingHttpMessageConverter

An HttpMessageConverter implementation that can read and write XML using Spring’s Marshaller and Unmarshaller abstractions from the org.springframework.oxm package. This converter requires a Marshaller and Unmarshaller before it can be used. These can be injected via constructor or bean properties. By default this converter supports ( text/xml) and ( application/xml).

MappingJackson2HttpMessageConverter

An HttpMessageConverter implementation that can read and write JSON using Jackson’s ObjectMapper. JSON mapping can be customized as needed through the use of Jackson’s provided annotations. When further control is needed, a custom ObjectMapper can be injected through the ObjectMapper property for cases where custom JSON serializers/deserializers need to be provided for specific types. By default this converter supports ( application/json).

MappingJackson2XmlHttpMessageConverter

An HttpMessageConverter implementation that can read and write XML using Jackson XML extension’s XmlMapper. XML mapping can be customized as needed through the use of JAXB or Jackson’s provided annotations. When further control is needed, a custom XmlMapper can be injected through the ObjectMapper property for cases where custom XML serializers/deserializers need to be provided for specific types. By default this converter supports ( application/xml).

SourceHttpMessageConverter

An HttpMessageConverter implementation that can read and write javax.xml.transform.Source from the HTTP request and response. Only DOMSource, SAXSource, and StreamSource are supported. By default, this converter supports ( text/xml) and ( application/xml).

BufferedImageHttpMessageConverter

An HttpMessageConverter implementation that can read and write java.awt.image.BufferedImage from the HTTP request and response. This converter reads and writes the media type supported by the Java I/O API.

Jackson JSON Views

It is possible to specify a Jackson JSON View to serialize only a subset of the object properties. For example:

MappingJacksonValue value = new MappingJacksonValue(new User("eric", "7!jd#h23"));

value.setSerializationView(User.WithoutPasswordView.class);

RequestEntity<MappingJacksonValue> requestEntity =

RequestEntity.post(new URI("http://example.com/user")).body(value);

ResponseEntity<String> response = template.exchange(requestEntity, String.class);

Multipart

To send multipart data, you need to provide a MultiValueMap<String, ?> whose values are either Objects representing part content, or HttpEntity representing the content and headers for a part. MultipartBodyBuilder provides a convenient API to prepare a multipart request:

MultipartBodyBuilder builder = new MultipartBodyBuilder();

builder.part("fieldPart", "fieldValue");

builder.part("filePart", new FileSystemResource("...logo.png"));

builder.part("jsonPart", new Person("Jason"));

MultiValueMap<String, HttpEntity<?>> parts = builder.build();

In most cases you do not have to specify the Content-Type for each part. The content type is determined automatically based on the HttpMessageConverter chosen to serialize it, or in the case of a Resource based on the file extension. If necessary you can explicitly provide the MediaType to use for each part through one fo the overloaded builder part methods.

Once the MultiValueMap is ready, you can pass it to the RestTemplate:

MultipartBodyBuilder builder = ...;

template.postForObject("http://example.com/upload", builder.build(), Void.class);

If the MultiValueMap contains at least one non-String value, which could also be represent regular form data (i.e. "application/x-www-form-urlencoded"), you don’t have to set the Content-Type to "multipart/form-data". This is always the case when using MultipartBodyBuilder which ensures an HttpEntity wrapper.

1.10.2. Async RestTemplate

The AsyncRestTemplate is deprecated. For all use cases where the AsyncRestTemplate is considered for use, please use the WebClient instead.

2. Enterprise JavaBeans (EJB) integration

2.1. Introduction

As a lightweight container, Spring is often considered an EJB replacement. We do believe that for many if not most applications and use cases, Spring as a container, combined with its rich supporting functionality in the area of transactions, ORM and JDBC access, is a better choice than implementing equivalent functionality via an EJB container and EJBs.

However, it is important to note that using Spring does not prevent you from using EJBs. In fact, Spring makes it much easier to access EJBs and implement EJBs and functionality within them. Additionally, using Spring to access services provided by EJBs allows the implementation of those services to later transparently be switched between local EJB, remote EJB, or POJO (plain old Java object) variants, without the client code having to be changed.

In this chapter, we look at how Spring can help you access and implement EJBs. Spring provides particular value when accessing stateless session beans (SLSBs), so we’ll begin by discussing this.

2.2. Accessing EJBs

2.2.1. Concepts

To invoke a method on a local or remote stateless session bean, client code must normally perform a JNDI lookup to obtain the (local or remote) EJB Home object, then use a 'create' method call on that object to obtain the actual (local or remote) EJB object. One or more methods are then invoked on the EJB.

To avoid repeated low-level code, many EJB applications use the Service Locator and Business Delegate patterns. These are better than spraying JNDI lookups throughout client code, but their usual implementations have significant disadvantages. For example:

Typically code using EJBs depends on Service Locator or Business Delegate singletons, making it hard to test.

In the case of the Service Locator pattern used without a Business Delegate, application code still ends up having to invoke the create() method on an EJB home, and deal with the resulting exceptions. Thus it remains tied to the EJB API and the complexity of the EJB programming model.

Implementing the Business Delegate pattern typically results in significant code duplication, where we have to write numerous methods that simply call the same method on the EJB.

The Spring approach is to allow the creation and use of proxy objects, normally configured inside a Spring container, which act as codeless business delegates. You do not need to write another Service Locator, another JNDI lookup, or duplicate methods in a hand-coded Business Delegate unless you are actually adding real value in such code.

2.2.2. Accessing local SLSBs

Assume that we have a web controller that needs to use a local EJB. We’ll follow best practice and use the EJB Business Methods Interface pattern, so that the EJB’s local interface extends a non EJB-specific business methods interface. Let’s call this business methods interface MyComponent.

public interface MyComponent {

...

}

One of the main reasons to use the Business Methods Interface pattern is to ensure that synchronization between method signatures in local interface and bean implementation class is automatic. Another reason is that it later makes it much easier for us to switch to a POJO (plain old Java object) implementation of the service if it makes sense to do so. Of course we’ll also need to implement the local home interface and provide an implementation class that implements SessionBean and the MyComponent business methods interface. Now the only Java coding we’ll need to do to hook up our web tier controller to the EJB implementation is to expose a setter method of type MyComponent on the controller. This will save the reference as an instance variable in the controller:

private MyComponent myComponent;

public void setMyComponent(MyComponent myComponent) {

this.myComponent = myComponent;

}

We can subsequently use this instance variable in any business method in the controller. Now assuming we are obtaining our controller object out of a Spring container, we can (in the same context) configure a LocalStatelessSessionProxyFactoryBean instance, which will be the EJB proxy object. The configuration of the proxy, and setting of the myComponent property of the controller is done with a configuration entry such as:

<bean id="myComponent"

class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">

<property name="jndiName" value="ejb/myBean"/>

<property name="businessInterface" value="com.mycom.MyComponent"/>

</bean>

<bean id="myController" class="com.mycom.myController">

<property name="myComponent" ref="myComponent"/>

</bean>

There’s a lot of work happening behind the scenes, courtesy of the Spring AOP framework, although you aren’t forced to work with AOP concepts to enjoy the results. The myComponent bean definition creates a proxy for the EJB, which implements the business method interface. The EJB local home is cached on startup, so there’s only a single JNDI lookup. Each time the EJB is invoked, the proxy invokes the classname method on the local EJB and invokes the corresponding business method on the EJB.

The myController bean definition sets the myComponent property of the controller class to the EJB proxy.

Alternatively (and preferably in case of many such proxy definitions), consider using the <jee:local-slsb> configuration element in Spring’s "jee" namespace:

<jee:local-slsb id="myComponent" jndi-name="ejb/myBean"

business-interface="com.mycom.MyComponent"/>

<bean id="myController" class="com.mycom.myController">

<property name="myComponent" ref="myComponent"/>

</bean>

This EJB access mechanism delivers huge simplification of application code: the web tier code (or other EJB client code) has no dependence on the use of EJB. If we want to replace this EJB reference with a POJO or a mock object or other test stub, we could simply change the myComponent bean definition without changing a line of Java code. Additionally, we haven’t had to write a single line of JNDI lookup or other EJB plumbing code as part of our application.

Benchmarks and experience in real applications indicate that the performance overhead of this approach (which involves reflective invocation of the target EJB) is minimal, and is typically undetectable in typical use. Remember that we don’t want to make fine-grained calls to EJBs anyway, as there’s a cost associated with the EJB infrastructure in the application server.

There is one caveat with regards to the JNDI lookup. In a bean container, this class is normally best used as a singleton (there simply is no reason to make it a prototype). However, if that bean container pre-instantiates singletons (as do the various XML ApplicationContext variants) you may have a problem if the bean container is loaded before the EJB container loads the target EJB. That is because the JNDI lookup will be performed in the init() method of this class and then cached, but the EJB will not have been bound at the target location yet. The solution is to not pre-instantiate this factory object, but allow it to be created on first use. In the XML containers, this is controlled via the lazy-init attribute.

Although this will not be of interest to the majority of Spring users, those doing programmatic AOP work with EJBs may want to look at LocalSlsbInvokerInterceptor.

2.2.3. Accessing remote SLSBs

Accessing remote EJBs is essentially identical to accessing local EJBs, except that the SimpleRemoteStatelessSessionProxyFactoryBean or <jee:remote-slsb> configuration element is used. Of course, with or without Spring, remote invocation semantics apply; a call to a method on an object in another VM in another computer does sometimes have to be treated differently in terms of usage scenarios and failure handling.

Spring’s EJB client support adds one more advantage over the non-Spring approach. Normally it is problematic for EJB client code to be easily switched back and forth between calling EJBs locally or remotely. This is because the remote interface methods must declare that they throw RemoteException, and client code must deal with this, while the local interface methods don’t. Client code written for local EJBs which needs to be moved to remote EJBs typically has to be modified to add handling for the remote exceptions, and client code written for remote EJBs which needs to be moved to local EJBs, can either stay the same but do a lot of unnecessary handling of remote exceptions, or needs to be modified to remove that code. With the Spring remote EJB proxy, you can instead not declare any thrown RemoteException in your Business Method Interface and implementing EJB code, have a remote interface which is identical except that it does throw RemoteException, and rely on the proxy to dynamically treat the two interfaces as if they were the same. That is, client code does not have to deal with the checked RemoteException class. Any actual RemoteException that is thrown during the EJB invocation will be re-thrown as the non-checked RemoteAccessException class, which is a subclass of RuntimeException. The target service can then be switched at will between a local EJB or remote EJB (or even plain Java object) implementation, without the client code knowing or caring. Of course, this is optional; there is nothing stopping you from declaring RemoteExceptions in your business interface.

2.2.4. Accessing EJB 2.x SLSBs versus EJB 3 SLSBs

Accessing EJB 2.x Session Beans and EJB 3 Session Beans via Spring is largely transparent. Spring’s EJB accessors, including the <jee:local-slsb> and <jee:remote-slsb> facilities, transparently adapt to the actual component at runtime. They handle a home interface if found (EJB 2.x style), or perform straight component invocations if no home interface is available (EJB 3 style).

Note: For EJB 3 Session Beans, you could effectively use a JndiObjectFactoryBean / <jee:jndi-lookup> as well, since fully usable component references are exposed for plain JNDI lookups there. Defining explicit <jee:local-slsb> / <jee:remote-slsb> lookups simply provides consistent and more explicit EJB access configuration.

3. JMS (Java Message Service)

3.1. Introduction

Spring provides a JMS integration framework that simplifies the use of the JMS API much like Spring’s integration does for the JDBC API.

JMS can be roughly divided into two areas of functionality, namely the production and consumption of messages. The JmsTemplate class is used for message production and synchronous message reception. For asynchronous reception similar to Java EE’s message-driven bean style, Spring provides a number of message listener containers that are used to create Message-Driven POJOs (MDPs). Spring also provides a declarative way of creating message listeners.

The package org.springframework.jms.core provides the core functionality for using JMS. It contains JMS template classes that simplify the use of the JMS by handling the creation and release of resources, much like the JdbcTemplate does for JDBC. The design principle common to Spring template classes is to provide helper methods to perform common operations and for more sophisticated usage, delegate the essence of the processing task to user implemented callback interfaces. The JMS template follows the same design. The classes offer various convenience methods for the sending of messages, consuming a message synchronously, and exposing the JMS session and message producer to the user.

The package org.springframework.jms.support provides JMSException translation functionality. The translation converts the checked JMSException hierarchy to a mirrored hierarchy of unchecked exceptions. If there are any provider specific subclasses of the checked javax.jms.JMSException, this exception is wrapped in the unchecked UncategorizedJmsException.

The package org.springframework.jms.support.converter provides a MessageConverter abstraction to convert between Java objects and JMS messages.

The package org.springframework.jms.support.destination provides various strategies for managing JMS destinations, such as providing a service locator for destinations stored in JNDI.

The package org.springframework.jms.annotation provides the necessary infrastructure to support annotation-driven listener endpoints using @JmsListener.

The package org.springframework.jms.config provides the parser implementation for the jms namespace as well the java config support to configure listener containers and create listener endpoints.

Finally, the package org.springframework.jms.connection provides an implementation of the ConnectionFactory suitable for use in standalone applications. It also contains an implementation of Spring’s PlatformTransactionManager for JMS (the cunningly named JmsTransactionManager). This allows for seamless integration of JMS as a transactional resource into Spring’s transaction management mechanisms.

3.2. Using Spring JMS

3.2.1. JmsTemplate

The JmsTemplate class is the central class in the JMS core package. It simplifies the use of JMS since it handles the creation and release of resources when sending or synchronously receiving messages.

Code that uses the JmsTemplate only needs to implement callback interfaces giving them a clearly defined high level contract. The MessageCreator callback interface creates a message given a Session provided by the calling code in JmsTemplate. In order to allow for more complex usage of the JMS API, the callback SessionCallback provides the user with the JMS session and the callback ProducerCallback exposes a Session and MessageProducer pair.

The JMS API exposes two types of send methods, one that takes delivery mode, priority, and time-to-live as Quality of Service (QOS) parameters and one that takes no QOS parameters which uses default values. Since there are many send methods in JmsTemplate, the setting of the QOS parameters have been exposed as bean properties to avoid duplication in the number of send methods. Similarly, the timeout value for synchronous receive calls is set using the property setReceiveTimeout.

Some JMS providers allow the setting of default QOS values administratively through the configuration of the ConnectionFactory. This has the effect that a call to MessageProducer's send method send(Destination destination, Message message) will use different QOS default values than those specified in the JMS specification. In order to provide consistent management of QOS values, the JmsTemplate must therefore be specifically enabled to use its own QOS values by setting the boolean property isExplicitQosEnabled to true.

For convenience, JmsTemplate also exposes a basic request-reply operation that allows to send a message and wait for a reply on a temporary queue that is created as part of the operation.

Instances of the JmsTemplate class are thread-safe once configured. This is important because it means that you can configure a single instance of a JmsTemplate and then safely inject this shared reference into multiple collaborators. To be clear, the JmsTemplate is stateful, in that it maintains a reference to a ConnectionFactory, but this state is not conversational state.

As of Spring Framework 4.1, JmsMessagingTemplate is built on top of JmsTemplate and provides an integration with the messaging abstraction, i.e. org.springframework.messaging.Message. This allows you to create the message to send in generic manner.

3.2.2. Connections

The JmsTemplate requires a reference to a ConnectionFactory. The ConnectionFactory is part of the JMS specification and serves as the entry point for working with JMS. It is used by the client application as a factory to create connections with the JMS provider and encapsulates various configuration parameters, many of which are vendor specific such as SSL configuration options.

When using JMS inside an EJB, the vendor provides implementations of the JMS interfaces so that they can participate in declarative transaction management and perform pooling of connections and sessions. In order to use this implementation, Java EE containers typically require that you declare a JMS connection factory as a resource-ref inside the EJB or servlet deployment descriptors. To ensure the use of these features with the JmsTemplate inside an EJB, the client application should ensure that it references the managed implementation of the ConnectionFactory.

Caching Messaging Resources

The standard API involves creating many intermediate objects. To send a message the following 'API' walk is performed

ConnectionFactory->Connection->Session->MessageProducer->send

Between the ConnectionFactory and the Send operation there are three intermediate objects that are created and destroyed. To optimise the resource usage and increase performance two implementations of ConnectionFactory are provided.

SingleConnectionFactory

Spring provides an implementation of the ConnectionFactory interface, SingleConnectionFactory, that will return the same Connection on all createConnection() calls and ignore calls to close(). This is useful for testing and standalone environments so that the same connection can be used for multiple JmsTemplate calls that may span any number of transactions. SingleConnectionFactory takes a reference to a standard ConnectionFactory that would typically come from JNDI.

CachingConnectionFactory

The CachingConnectionFactory extends the functionality of SingleConnectionFactory and adds the caching of Sessions, MessageProducers, and MessageConsumers. The initial cache size is set to 1, use the property sessionCacheSize to increase the number of cached sessions. Note that the number of actual cached sessions will be more than that number as sessions are cached based on their acknowledgment mode, so there can be up to 4 cached session instances when sessionCacheSize is set to one, one for each acknowledgment mode. MessageProducers and MessageConsumers are cached within their owning session and also take into account the unique properties of the producers and consumers when caching. MessageProducers are cached based on their destination. MessageConsumers are cached based on a key composed of the destination, selector, noLocal delivery flag, and the durable subscription name (if creating durable consumers).

3.2.3. Destination Management

Destinations, like ConnectionFactories, are JMS administered objects that can be stored and retrieved in JNDI. When configuring a Spring application context you can use the JNDI factory class JndiObjectFactoryBean / <jee:jndi-lookup> to perform dependency injection on your object’s references to JMS destinations. However, often this strategy is cumbersome if there are a large number of destinations in the application or if there are advanced destination management features unique to the JMS provider. Examples of such advanced destination management would be the creation of dynamic destinations or support for a hierarchical namespace of destinations. The JmsTemplate delegates the resolution of a destination name to a JMS destination object to an implementation of the interface DestinationResolver. DynamicDestinationResolver is the default implementation used by JmsTemplate and accommodates resolving dynamic destinations. A JndiDestinationResolver is also provided that acts as a service locator for destinations contained in JNDI and optionally falls back to the behavior contained in DynamicDestinationResolver.

Quite often the destinations used in a JMS application are only known at runtime and therefore cannot be administratively created when the application is deployed. This is often because there is shared application logic between interacting system components that create destinations at runtime according to a well-known naming convention. Even though the creation of dynamic destinations is not part of the JMS specification, most vendors have provided this functionality. Dynamic destinations are created with a name defined by the user which differentiates them from temporary destinations and are often not registered in JNDI. The API used to create dynamic destinations varies from provider to provider since the properties associated with the destination are vendor specific. However, a simple implementation choice that is sometimes made by vendors is to disregard the warnings in the JMS specification and to use the TopicSession method createTopic(String topicName) or the QueueSession method createQueue(String queueName) to create a new destination with default destination properties. Depending on the vendor implementation, DynamicDestinationResolver may then also create a physical destination instead of only resolving one.

The boolean property pubSubDomain is used to configure the JmsTemplate with knowledge of what JMS domain is being used. By default the value of this property is false, indicating that the point-to-point domain, Queues, will be used. This property used by JmsTemplate determines the behavior of dynamic destination resolution via implementations of the DestinationResolver interface.

You can also configure the JmsTemplate with a default destination via the property defaultDestination. The default destination will be used with send and receive operations that do not refer to a specific destination.

3.2.4. Message Listener Containers

One of the most common uses of JMS messages in the EJB world is to drive message-driven beans (MDBs). Spring offers a solution to create message-driven POJOs (MDPs) in a way that does not tie a user to an EJB container. (See Asynchronous reception: Message-Driven POJOs for detailed coverage of Spring’s MDP support.) As from Spring Framework 4.1, endpoint methods can be simply annotated using @JmsListener see Annotation-driven listener endpoints for more details.

A message listener container is used to receive messages from a JMS message queue and drive the MessageListener that is injected into it. The listener container is responsible for all threading of message reception and dispatches into the listener for processing. A message listener container is the intermediary between an MDP and a messaging provider, and takes care of registering to receive messages, participating in transactions, resource acquisition and release, exception conversion and suchlike. This allows you as an application developer to write the (possibly complex) business logic associated with receiving a message (and possibly responding to it), and delegates boilerplate JMS infrastructure concerns to the framework.

There are two standard JMS message listener containers packaged with Spring, each with its specialised feature set.

SimpleMessageListenerContainer

This message listener container is the simpler of the two standard flavors. It creates a fixed number of JMS sessions and consumers at startup, registers the listener using the standard JMS MessageConsumer.setMessageListener() method, and leaves it up the JMS provider to perform listener callbacks. This variant does not allow for dynamic adaption to runtime demands or for participation in externally managed transactions. Compatibility-wise, it stays very close to the spirit of the standalone JMS specification - but is generally not compatible with Java EE’s JMS restrictions.

While SimpleMessageListenerContainer does not allow for the participation in externally managed transactions, it does support native JMS transactions: simply switch the 'sessionTransacted' flag to 'true' or, in the namespace, set the 'acknowledge' attribute to 'transacted': Exceptions thrown from your listener will lead to a rollback then, with the message getting redelivered. Alternatively, consider using 'CLIENT\_ACKNOWLEDGE' mode which provides redelivery in case of an exception as well but does not use transacted Sessions and therefore does not include any other Session operations (such as sending response messages) in the transaction protocol.

The default 'AUTO\_ACKNOWLEDGE' mode does not provide proper reliability guarantees. Messages may get lost when listener execution fails (since the provider will automatically acknowledge each message after listener invocation, with no exceptions to be propagated to the provider) or when the listener container shuts down (this may be configured through the 'acceptMessagesWhileStopping' flag). Make sure to use transacted sessions in case of reliability needs, e.g. for reliable queue handling and durable topic subscriptions.

DefaultMessageListenerContainer

This message listener container is the one used in most cases. In contrast to SimpleMessageListenerContainer, this container variant allows for dynamic adaptation to runtime demands and is able to participate in externally managed transactions. Each received message is registered with an XA transaction when configured with a JtaTransactionManager; so processing may take advantage of XA transaction semantics. This listener container strikes a good balance between low requirements on the JMS provider, advanced functionality such as the participation in externally managed transactions, and compatibility with Java EE environments.

The cache level of the container can be customized. Note that when no caching is enabled, a new connection and a new session is created for each message reception. Combining this with a non durable subscription with high loads may lead to message lost. Make sure to use a proper cache level in such case.

This container also has recoverable capabilities when the broker goes down. By default, a simple BackOff implementation retries every 5 seconds. It is possible to specify a custom BackOff implementation for more fine-grained recovery options, see ExponentialBackOff for an example.

Like its sibling SimpleMessageListenerContainer, DefaultMessageListenerContainer supports native JMS transactions and also allows for customizing the acknowledgment mode. This is strongly recommended over externally managed transactions if feasible for your scenario: that is, if you can live with occasional duplicate messages in case of the JVM dying. Custom duplicate message detection steps in your business logic may cover such situations, e.g. in the form of a business entity existence check or a protocol table check. Any such arrangements will be significantly more efficient than the alternative: wrapping your entire processing with an XA transaction (through configuring your DefaultMessageListenerContainer with an JtaTransactionManager), covering the reception of the JMS message as well as the execution of the business logic in your message listener (including database operations etc).

The default 'AUTO\_ACKNOWLEDGE' mode does not provide proper reliability guarantees. Messages may get lost when listener execution fails (since the provider will automatically acknowledge each message before listener invocation) or when the listener container shuts down (this may be configured through the 'acceptMessagesWhileStopping' flag). Make sure to use transacted sessions in case of reliability needs, e.g. for reliable queue handling and durable topic subscriptions.

3.2.5. Transaction management

Spring provides a JmsTransactionManager that manages transactions for a single JMS ConnectionFactory. This allows JMS applications to leverage the managed transaction features of Spring as described in Transaction Management. The JmsTransactionManager performs local resource transactions, binding a JMS Connection/Session pair from the specified ConnectionFactory to the thread. JmsTemplate automatically detects such transactional resources and operates on them accordingly.

In a Java EE environment, the ConnectionFactory will pool Connections and Sessions, so those resources are efficiently reused across transactions. In a standalone environment, using Spring’s SingleConnectionFactory will result in a shared JMS Connection, with each transaction having its own independent Session. Alternatively, consider the use of a provider-specific pooling adapter such as ActiveMQ’s PooledConnectionFactory class.

JmsTemplate can also be used with the JtaTransactionManager and an XA-capable JMS ConnectionFactory for performing distributed transactions. Note that this requires the use of a JTA transaction manager as well as a properly XA-configured ConnectionFactory! (Check your Java EE server’s / JMS provider’s documentation.)

Reusing code across a managed and unmanaged transactional environment can be confusing when using the JMS API to create a Session from a Connection. This is because the JMS API has only one factory method to create a Session and it requires values for the transaction and acknowledgment modes. In a managed environment, setting these values is the responsibility of the environment’s transactional infrastructure, so these values are ignored by the vendor’s wrapper to the JMS Connection. When using the JmsTemplate in an unmanaged environment you can specify these values through the use of the properties sessionTransacted and sessionAcknowledgeMode. When using a PlatformTransactionManager with JmsTemplate, the template will always be given a transactional JMS Session.

3.3. Sending a Message

The JmsTemplate contains many convenience methods to send a message. There are send methods that specify the destination using a javax.jms.Destination object and those that specify the destination using a string for use in a JNDI lookup. The send method that takes no destination argument uses the default destination.

import javax.jms.ConnectionFactory;

import javax.jms.JMSException;

import javax.jms.Message;

import javax.jms.Queue;

import javax.jms.Session;

import org.springframework.jms.core.MessageCreator;

import org.springframework.jms.core.JmsTemplate;

public class JmsQueueSender {

private JmsTemplate jmsTemplate;

private Queue queue;

public void setConnectionFactory(ConnectionFactory cf) {

this.jmsTemplate = new JmsTemplate(cf);

}

public void setQueue(Queue queue) {

this.queue = queue;

}

public void simpleSend() {

this.jmsTemplate.send(this.queue, new MessageCreator() {

public Message createMessage(Session session) throws JMSException {

return session.createTextMessage("hello queue world");

}

});

}

}

This example uses the MessageCreator callback to create a text message from the supplied Session object. The JmsTemplate is constructed by passing a reference to a ConnectionFactory. As an alternative, a zero argument constructor and connectionFactory is provided and can be used for constructing the instance in JavaBean style (using a BeanFactory or plain Java code). Alternatively, consider deriving from Spring’s JmsGatewaySupport convenience base class, which provides pre-built bean properties for JMS configuration.

The method send(String destinationName, MessageCreator creator) lets you send a message using the string name of the destination. If these names are registered in JNDI, you should set the destinationResolver property of the template to an instance of JndiDestinationResolver.

If you created the JmsTemplate and specified a default destination, the send(MessageCreator c) sends a message to that destination.

3.3.1. Using Message Converters

In order to facilitate the sending of domain model objects, the JmsTemplate has various send methods that take a Java object as an argument for a message’s data content. The overloaded methods convertAndSend() and receiveAndConvert() in JmsTemplate delegate the conversion process to an instance of the MessageConverter interface. This interface defines a simple contract to convert between Java objects and JMS messages. The default implementation SimpleMessageConverter supports conversion between String and TextMessage, byte[] and BytesMesssage, and java.util.Map and MapMessage. By using the converter, you and your application code can focus on the business object that is being sent or received via JMS and not be concerned with the details of how it is represented as a JMS message.

The sandbox currently includes a MapMessageConverter which uses reflection to convert between a JavaBean and a MapMessage. Other popular implementation choices you might implement yourself are Converters that use an existing XML marshalling package, such as JAXB, Castor or XStream, to create a TextMessage representing the object.

To accommodate the setting of a message’s properties, headers, and body that can not be generically encapsulated inside a converter class, the MessagePostProcessor interface gives you access to the message after it has been converted, but before it is sent. The example below demonstrates how to modify a message header and a property after a java.util.Map is converted to a message.

public void sendWithConversion() {

Map map = new HashMap();

map.put("Name", "Mark");

map.put("Age", new Integer(47));

jmsTemplate.convertAndSend("testQueue", map, new MessagePostProcessor() {

public Message postProcessMessage(Message message) throws JMSException {

message.setIntProperty("AccountID", 1234);

message.setJMSCorrelationID("123-00001");

return message;

}

});

}

This results in a message of the form:

MapMessage={

Header={

... standard headers ...

CorrelationID={123-00001}

}

Properties={

AccountID={Integer:1234}

}

Fields={

Name={String:Mark}

Age={Integer:47}

}

}

3.3.2. SessionCallback and ProducerCallback

While the send operations cover many common usage scenarios, there are cases when you want to perform multiple operations on a JMS Session or MessageProducer. The SessionCallback and ProducerCallback expose the JMS Session and Session / MessageProducer pair respectively. The execute() methods on JmsTemplate execute these callback methods.

3.4. Receiving a message

3.4.1. Synchronous reception

While JMS is typically associated with asynchronous processing, it is possible to consume messages synchronously. The overloaded receive(..) methods provide this functionality. During a synchronous receive, the calling thread blocks until a message becomes available. This can be a dangerous operation since the calling thread can potentially be blocked indefinitely. The property receiveTimeout specifies how long the receiver should wait before giving up waiting for a message.

3.4.2. Asynchronous reception: Message-Driven POJOs

Spring also supports annotated-listener endpoints through the use of the @JmsListener annotation and provides an open infrastructure to register endpoints programmatically. This is by far the most convenient way to setup an asynchronous receiver, see Enable listener endpoint annotations for more details.

In a fashion similar to a Message-Driven Bean (MDB) in the EJB world, the Message-Driven POJO (MDP) acts as a receiver for JMS messages. The one restriction (but see also below for the discussion of the MessageListenerAdapter class) on an MDP is that it must implement the javax.jms.MessageListener interface. Please also be aware that in the case where your POJO will be receiving messages on multiple threads, it is important to ensure that your implementation is thread-safe.

Below is a simple implementation of an MDP:

import javax.jms.JMSException;

import javax.jms.Message;

import javax.jms.MessageListener;

import javax.jms.TextMessage;

public class ExampleListener implements MessageListener {

public void onMessage(Message message) {

if (message instanceof TextMessage) {

try {

System.out.println(((TextMessage) message).getText());

}

catch (JMSException ex) {

throw new RuntimeException(ex);

}

}

else {

throw new IllegalArgumentException("Message must be of type TextMessage");

}

}

}

Once you’ve implemented your MessageListener, it’s time to create a message listener container.

Find below an example of how to define and configure one of the message listener containers that ships with Spring (in this case the DefaultMessageListenerContainer).

<!-- this is the Message Driven POJO (MDP) -->

<bean id="messageListener" class="jmsexample.ExampleListener"/>

<!-- and this is the message listener container -->

<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">

<property name="connectionFactory" ref="connectionFactory"/>

<property name="destination" ref="destination"/>

<property name="messageListener" ref="messageListener"/>

</bean>

Please refer to the Spring javadocs of the various message listener containers for a full description of the features supported by each implementation.

3.4.3. SessionAwareMessageListener interface

The SessionAwareMessageListener interface is a Spring-specific interface that provides a similar contract to the JMS MessageListener interface, but also provides the message handling method with access to the JMS Session from which the Message was received.

package org.springframework.jms.listener;

public interface SessionAwareMessageListener {

void onMessage(Message message, Session session) throws JMSException;

}

You can choose to have your MDPs implement this interface (in preference to the standard JMS MessageListener interface) if you want your MDPs to be able to respond to any received messages (using the Session supplied in the onMessage(Message, Session) method). All of the message listener container implementations that ship with Spring have support for MDPs that implement either the MessageListener or SessionAwareMessageListener interface. Classes that implement the SessionAwareMessageListener come with the caveat that they are then tied to Spring through the interface. The choice of whether or not to use it is left entirely up to you as an application developer or architect.

Please note that the 'onMessage(..)' method of the SessionAwareMessageListener interface throws JMSException. In contrast to the standard JMS MessageListener interface, when using the SessionAwareMessageListener interface, it is the responsibility of the client code to handle any exceptions thrown.

3.4.4. MessageListenerAdapter

The MessageListenerAdapter class is the final component in Spring’s asynchronous messaging support: in a nutshell, it allows you to expose almost any class as a MDP (there are of course some constraints).

Consider the following interface definition. Notice that although the interface extends neither the MessageListener nor SessionAwareMessageListener interfaces, it can still be used as a MDP via the use of the MessageListenerAdapter class. Notice also how the various message handling methods are strongly typed according to the contents of the various Message types that they can receive and handle.

public interface MessageDelegate {

void handleMessage(String message);

void handleMessage(Map message);

void handleMessage(byte[] message);

void handleMessage(Serializable message);

}

public class DefaultMessageDelegate implements MessageDelegate {

// implementation elided for clarity...

}

In particular, note how the above implementation of the MessageDelegate interface (the above DefaultMessageDelegate class) has no JMS dependencies at all. It truly is a POJO that we will make into an MDP via the following configuration.

<!-- this is the Message Driven POJO (MDP) -->

<bean id="messageListener" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">

<constructor-arg>

<bean class="jmsexample.DefaultMessageDelegate"/>

</constructor-arg>

</bean>

<!-- and this is the message listener container... -->

<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">

<property name="connectionFactory" ref="connectionFactory"/>

<property name="destination" ref="destination"/>

<property name="messageListener" ref="messageListener"/>

</bean>

Below is an example of another MDP that can only handle the receiving of JMS TextMessage messages. Notice how the message handling method is actually called 'receive' (the name of the message handling method in a MessageListenerAdapter defaults to 'handleMessage'), but it is configurable (as you will see below). Notice also how the 'receive(..)' method is strongly typed to receive and respond only to JMS TextMessage messages.

public interface TextMessageDelegate {

void receive(TextMessage message);

}

public class DefaultTextMessageDelegate implements TextMessageDelegate {

// implementation elided for clarity...

}

The configuration of the attendant MessageListenerAdapter would look like this:

<bean id="messageListener" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">

<constructor-arg>

<bean class="jmsexample.DefaultTextMessageDelegate"/>

</constructor-arg>

<property name="defaultListenerMethod" value="receive"/>

<!-- we don't want automatic message context extraction -->

<property name="messageConverter">

<null/>

</property>

</bean>

Please note that if the above 'messageListener' receives a JMS Message of a type other than TextMessage, an IllegalStateException will be thrown (and subsequently swallowed). Another of the capabilities of the MessageListenerAdapter class is the ability to automatically send back a response Message if a handler method returns a non-void value. Consider the interface and class:

public interface ResponsiveTextMessageDelegate {

// notice the return type...

String receive(TextMessage message);

}

public class DefaultResponsiveTextMessageDelegate implements ResponsiveTextMessageDelegate {

// implementation elided for clarity...

}

If the above DefaultResponsiveTextMessageDelegate is used in conjunction with a MessageListenerAdapter then any non-null value that is returned from the execution of the 'receive(..)' method will (in the default configuration) be converted into a TextMessage. The resulting TextMessage will then be sent to the Destination (if one exists) defined in the JMS Reply-To property of the original Message, or the default Destination set on the MessageListenerAdapter (if one has been configured); if no Destination is found then an InvalidDestinationException will be thrown (and please note that this exception will not be swallowed and will propagate up the call stack).

3.4.5. Processing messages within transactions

Invoking a message listener within a transaction only requires reconfiguration of the listener container.

Local resource transactions can simply be activated through the sessionTransacted flag on the listener container definition. Each message listener invocation will then operate within an active JMS transaction, with message reception rolled back in case of listener execution failure. Sending a response message (via SessionAwareMessageListener) will be part of the same local transaction, but any other resource operations (such as database access) will operate independently. This usually requires duplicate message detection in the listener implementation, covering the case where database processing has committed but message processing failed to commit.

<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">

<property name="connectionFactory" ref="connectionFactory"/>

<property name="destination" ref="destination"/>

<property name="messageListener" ref="messageListener"/>

<property name="sessionTransacted" value="true"/>

</bean>

For participating in an externally managed transaction, you will need to configure a transaction manager and use a listener container which supports externally managed transactions: typically DefaultMessageListenerContainer.

To configure a message listener container for XA transaction participation, you’ll want to configure a JtaTransactionManager (which, by default, delegates to the Java EE server’s transaction subsystem). Note that the underlying JMS ConnectionFactory needs to be XA-capable and properly registered with your JTA transaction coordinator! (Check your Java EE server’s configuration of JNDI resources.) This allows message reception as well as e.g. database access to be part of the same transaction (with unified commit semantics, at the expense of XA transaction log overhead).

<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>

Then you just need to add it to our earlier container configuration. The container will take care of the rest.

<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">

<property name="connectionFactory" ref="connectionFactory"/>

<property name="destination" ref="destination"/>

<property name="messageListener" ref="messageListener"/>

<property name="transactionManager" ref="transactionManager"/>

</bean>

3.5. Support for JCA Message Endpoints

Beginning with version 2.5, Spring also provides support for a JCA-based MessageListener container. The JmsMessageEndpointManager will attempt to automatically determine the ActivationSpec class name from the provider’s ResourceAdapter class name. Therefore, it is typically possible to just provide Spring’s generic JmsActivationSpecConfig as shown in the following example.

<bean class="org.springframework.jms.listener.endpoint.JmsMessageEndpointManager">

<property name="resourceAdapter" ref="resourceAdapter"/>

<property name="activationSpecConfig">

<bean class="org.springframework.jms.listener.endpoint.JmsActivationSpecConfig">

<property name="destinationName" value="myQueue"/>

</bean>

</property>

<property name="messageListener" ref="myMessageListener"/>

</bean>

Alternatively, you may set up a JmsMessageEndpointManager with a given ActivationSpec object. The ActivationSpec object may also come from a JNDI lookup (using <jee:jndi-lookup>).

<bean class="org.springframework.jms.listener.endpoint.JmsMessageEndpointManager">

<property name="resourceAdapter" ref="resourceAdapter"/>

<property name="activationSpec">

<bean class="org.apache.activemq.ra.ActiveMQActivationSpec">

<property name="destination" value="myQueue"/>

<property name="destinationType" value="javax.jms.Queue"/>

</bean>

</property>

<property name="messageListener" ref="myMessageListener"/>

</bean>

Using Spring’s ResourceAdapterFactoryBean, the target ResourceAdapter may be configured locally as depicted in the following example.

<bean id="resourceAdapter" class="org.springframework.jca.support.ResourceAdapterFactoryBean">

<property name="resourceAdapter">

<bean class="org.apache.activemq.ra.ActiveMQResourceAdapter">

<property name="serverUrl" value="tcp://localhost:61616"/>

</bean>

</property>

<property name="workManager">

<bean class="org.springframework.jca.work.SimpleTaskWorkManager"/>

</property>

</bean>

The specified WorkManager may also point to an environment-specific thread pool - typically through SimpleTaskWorkManager's "asyncTaskExecutor" property. Consider defining a shared thread pool for all your ResourceAdapter instances if you happen to use multiple adapters.

In some environments (e.g. WebLogic 9 or above), the entire ResourceAdapter object may be obtained from JNDI instead (using <jee:jndi-lookup>). The Spring-based message listeners can then interact with the server-hosted ResourceAdapter, also using the server’s built-in WorkManager.

Please consult the javadoc for JmsMessageEndpointManager, JmsActivationSpecConfig, and ResourceAdapterFactoryBean for more details.

Spring also provides a generic JCA message endpoint manager which is not tied to JMS: org.springframework.jca.endpoint.GenericMessageEndpointManager. This component allows for using any message listener type (e.g. a CCI MessageListener) and any provider-specific ActivationSpec object. Check out your JCA provider’s documentation to find out about the actual capabilities of your connector, and consult GenericMessageEndpointManager's javadoc for the Spring-specific configuration details.

JCA-based message endpoint management is very analogous to EJB 2.1 Message-Driven Beans; it uses the same underlying resource provider contract. Like with EJB 2.1 MDBs, any message listener interface supported by your JCA provider can be used in the Spring context as well. Spring nevertheless provides explicit 'convenience' support for JMS, simply because JMS is the most common endpoint API used with the JCA endpoint management contract.

3.6. Annotation-driven listener endpoints

The easiest way to receive a message asynchronously is to use the annotated listener endpoint infrastructure. In a nutshell, it allows you to expose a method of a managed bean as a JMS listener endpoint.

@Component

public class MyService {

@JmsListener(destination = "myDestination")

public void processOrder(String data) { ... }

}

The idea of the example above is that whenever a message is available on the javax.jms.Destination "myDestination", the processOrder method is invoked accordingly (in this case, with the content of the JMS message similarly to what the MessageListenerAdapter provides).

The annotated endpoint infrastructure creates a message listener container behind the scenes for each annotated method, using a JmsListenerContainerFactory. Such a container is not registered against the application context but can be easily located for management purposes using the JmsListenerEndpointRegistry bean.

@JmsListener is a repeatable annotation on Java 8, so it is possible to associate several JMS destinations to the same method by adding additional @JmsListener declarations to it.

3.6.1. Enable listener endpoint annotations

To enable support for @JmsListener annotations add @EnableJms to one of your @Configuration classes.

@Configuration

@EnableJms

public class AppConfig {

@Bean

public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() {

DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();

factory.setConnectionFactory(connectionFactory());

factory.setDestinationResolver(destinationResolver());

factory.setSessionTransacted(true);

factory.setConcurrency("3-10");

return factory;

}

}

By default, the infrastructure looks for a bean named jmsListenerContainerFactory as the source for the factory to use to create message listener containers. In this case, and ignoring the JMS infrastructure setup, the processOrder method can be invoked with a core poll size of 3 threads and a maximum pool size of 10 threads.

It is possible to customize the listener container factory to use per annotation or an explicit default can be configured by implementing the JmsListenerConfigurer interface. The default is only required if at least one endpoint is registered without a specific container factory. See the javadoc for full details and examples.

If you prefer XML configuration use the <jms:annotation-driven> element.

<jms:annotation-driven/>

<bean id="jmsListenerContainerFactory"

class="org.springframework.jms.config.DefaultJmsListenerContainerFactory">

<property name="connectionFactory" ref="connectionFactory"/>

<property name="destinationResolver" ref="destinationResolver"/>

<property name="sessionTransacted" value="true"/>

<property name="concurrency" value="3-10"/>

</bean>

3.6.2. Programmatic endpoints registration

JmsListenerEndpoint provides a model of an JMS endpoint and is responsible for configuring the container for that model. The infrastructure allows you to configure endpoints programmatically in addition to the ones that are detected by the JmsListener annotation.

@Configuration

@EnableJms

public class AppConfig implements JmsListenerConfigurer {

@Override

public void configureJmsListeners(JmsListenerEndpointRegistrar registrar) {

SimpleJmsListenerEndpoint endpoint = new SimpleJmsListenerEndpoint();

endpoint.setId("myJmsEndpoint");

endpoint.setDestination("anotherQueue");

endpoint.setMessageListener(message -> {

// processing

});

registrar.registerEndpoint(endpoint);

}

}

In the example above, we used SimpleJmsListenerEndpoint which provides the actual MessageListener to invoke but you could just as well build your own endpoint variant describing a custom invocation mechanism.

It should be noted that you could just as well skip the use of @JmsListener altogether and only register your endpoints programmatically through JmsListenerConfigurer.

3.6.3. Annotated endpoint method signature

So far, we have been injecting a simple String in our endpoint but it can actually have a very flexible method signature. Let’s rewrite it to inject the Order with a custom header:

@Component

public class MyService {

@JmsListener(destination = "myDestination")

public void processOrder(Order order, @Header("order\_type") String orderType) {

...

}

}

These are the main elements you can inject in JMS listener endpoints:

The raw javax.jms.Message or any of its subclasses (provided of course that it matches the incoming message type).

The javax.jms.Session for optional access to the native JMS API e.g. for sending a custom reply.

The org.springframework.messaging.Message representing the incoming JMS message. Note that this message holds both the custom and the standard headers (as defined by JmsHeaders).

@Header-annotated method arguments to extract a specific header value, including standard JMS headers.

@Headers-annotated argument that must also be assignable to java.util.Map for getting access to all headers.

A non-annotated element that is not one of the supported types (i.e. Message and Session) is considered to be the payload. You can make that explicit by annotating the parameter with @Payload. You can also turn on validation by adding an extra @Valid.

The ability to inject Spring’s Message abstraction is particularly useful to benefit from all the information stored in the transport-specific message without relying on transport-specific API.

@JmsListener(destination = "myDestination")

public void processOrder(Message<Order> order) { ... }

Handling of method arguments is provided by DefaultMessageHandlerMethodFactory which can be further customized to support additional method arguments. The conversion and validation support can be customized there as well.

For instance, if we want to make sure our Order is valid before processing it, we can annotate the payload with @Valid and configure the necessary validator as follows:

@Configuration

@EnableJms

public class AppConfig implements JmsListenerConfigurer {

@Override

public void configureJmsListeners(JmsListenerEndpointRegistrar registrar) {

registrar.setMessageHandlerMethodFactory(myJmsHandlerMethodFactory());

}

@Bean

public DefaultMessageHandlerMethodFactory myHandlerMethodFactory() {

DefaultMessageHandlerMethodFactory factory = new DefaultMessageHandlerMethodFactory();

factory.setValidator(myValidator());

return factory;

}

}

3.6.4. Response management

The existing support in MessageListenerAdapter already allows your method to have a non-void return type. When that’s the case, the result of the invocation is encapsulated in a javax.jms.Message sent either in the destination specified in the JMSReplyTo header of the original message or in the default destination configured on the listener. That default destination can now be set using the @SendTo annotation of the messaging abstraction.

Assuming our processOrder method should now return an OrderStatus, it is possible to write it as follow to automatically send a response:

@JmsListener(destination = "myDestination")

@SendTo("status")

public OrderStatus processOrder(Order order) {

// order processing

return status;

}

If you have several @JmsListener-annotated methods, you can also place the @SendTo annotation at the class level to share a default reply destination.

If you need to set additional headers in a transport-independent manner, you could return a Message instead, something like:

@JmsListener(destination = "myDestination")

@SendTo("status")

public Message<OrderStatus> processOrder(Order order) {

// order processing

return MessageBuilder

.withPayload(status)

.setHeader("code", 1234)

.build();

}

If you need to compute the response destination at runtime, you can encapsulate your response in a JmsResponse instance that also provides the destination to use at runtime. The previous example can be rewritten as follows:

@JmsListener(destination = "myDestination")

public JmsResponse<Message<OrderStatus>> processOrder(Order order) {

// order processing

Message<OrderStatus> response = MessageBuilder

.withPayload(status)

.setHeader("code", 1234)

.build();

return JmsResponse.forQueue(response, "status");

}

Finally if you need to specify some QoS values for the response such as the priority or the time to live, you can configure the JmsListenerContainerFactory accordingly:

@Configuration

@EnableJms

public class AppConfig {

@Bean

public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() {

DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();

factory.setConnectionFactory(connectionFactory());

QosSettings replyQosSettings = new QosSettings();

replyQosSettings.setPriority(2);

replyQosSettings.setTimeToLive(10000);

factory.setReplyQosSettings(replyQosSettings);

return factory;

}

}

3.7. JMS namespace support

Spring provides an XML namespace for simplifying JMS configuration. To use the JMS namespace elements you will need to reference the JMS schema:

<?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:jms="http://www.springframework.org/schema/jms"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms.xsd">

<!-- bean definitions here -->

</beans>

The namespace consists of three top-level elements: <annotation-driven/>, <listener-container/> and <jca-listener-container/>. <annotation-driven enables the use of annotation-driven listener endpoints. <listener-container/> and <jca-listener-container/> defines shared listener container configuration and may contain <listener/> child elements. Here is an example of a basic configuration for two listeners.

<jms:listener-container>

<jms:listener destination="queue.orders" ref="orderService" method="placeOrder"/>

<jms:listener destination="queue.confirmations" ref="confirmationLogger" method="log"/>

</jms:listener-container>

The example above is equivalent to creating two distinct listener container bean definitions and two distinct MessageListenerAdapter bean definitions as demonstrated in MessageListenerAdapter. In addition to the attributes shown above, the listener element may contain several optional ones. The following table describes all available attributes:

Table 3. Attributes of the JMS <listener> element

Attribute Description

id

A bean name for the hosting listener container. If not specified, a bean name will be automatically generated.

destination (required)

The destination name for this listener, resolved through the DestinationResolver strategy.

ref (required)

The bean name of the handler object.

method

The name of the handler method to invoke. If the ref points to a MessageListener or Spring SessionAwareMessageListener, this attribute may be omitted.

response-destination

The name of the default response destination to send response messages to. This will be applied in case of a request message that does not carry a "JMSReplyTo" field. The type of this destination will be determined by the listener-container’s "response-destination-type" attribute. Note: This only applies to a listener method with a return value, for which each result object will be converted into a response message.

subscription

The name of the durable subscription, if any.

selector

An optional message selector for this listener.

concurrency

The number of concurrent sessions/consumers to start for this listener. Can either be a simple number indicating the maximum number (e.g. "5") or a range indicating the lower as well as the upper limit (e.g. "3-5"). Note that a specified minimum is just a hint and might be ignored at runtime. Default is the value provided by the container

The <listener-container/> element also accepts several optional attributes. This allows for customization of the various strategies (for example, taskExecutor and destinationResolver) as well as basic JMS settings and resource references. Using these attributes, it is possible to define highly-customized listener containers while still benefiting from the convenience of the namespace.

Such settings can be automatically exposed as a JmsListenerContainerFactory by specifying the id of the bean to expose through the factory-id attribute.

<jms:listener-container connection-factory="myConnectionFactory"

task-executor="myTaskExecutor"

destination-resolver="myDestinationResolver"

transaction-manager="myTransactionManager"

concurrency="10">

<jms:listener destination="queue.orders" ref="orderService" method="placeOrder"/>

<jms:listener destination="queue.confirmations" ref="confirmationLogger" method="log"/>

</jms:listener-container>

The following table describes all available attributes. Consult the class-level javadocs of the AbstractMessageListenerContainer and its concrete subclasses for more details on the individual properties. The javadocs also provide a discussion of transaction choices and message redelivery scenarios.

Table 4. Attributes of the JMS <listener-container> element

Attribute Description

container-type

The type of this listener container. Available options are: default, simple, default102, or simple102 (the default value is 'default').

container-class

A custom listener container implementation class as fully qualified class name. Default is Spring’s standard DefaultMessageListenerContainer or SimpleMessageListenerContainer, according to the "container-type" attribute.

factory-id

Exposes the settings defined by this element as a JmsListenerContainerFactory with the specified id so that they can be reused with other endpoints.

connection-factory

A reference to the JMS ConnectionFactory bean (the default bean name is 'connectionFactory').

task-executor

A reference to the Spring TaskExecutor for the JMS listener invokers.

destination-resolver

A reference to the DestinationResolver strategy for resolving JMS Destinations.

message-converter

A reference to the MessageConverter strategy for converting JMS Messages to listener method arguments. Default is a SimpleMessageConverter.

error-handler

A reference to an ErrorHandler strategy for handling any uncaught Exceptions that may occur during the execution of the MessageListener.

destination-type

The JMS destination type for this listener: queue, topic, durableTopic, sharedTopic or sharedDurableTopic. This enables potentially the pubSubDomain, subscriptionDurable and subscriptionShared properties of the container. The default is queue (i.e. disabling those 3 properties).

response-destination-type

The JMS destination type for responses: "queue", "topic". Default is the value of the "destination-type" attribute.

client-id

The JMS client id for this listener container. Needs to be specified when using durable subscriptions.

cache

The cache level for JMS resources: none, connection, session, consumer or auto. By default ( auto), the cache level will effectively be "consumer", unless an external transaction manager has been specified - in which case the effective default will be none (assuming Java EE-style transaction management where the given ConnectionFactory is an XA-aware pool).

acknowledge

The native JMS acknowledge mode: auto, client, dups-ok or transacted. A value of transacted activates a locally transacted Session. As an alternative, specify the transaction-manager attribute described below. Default is auto.

transaction-manager

A reference to an external PlatformTransactionManager (typically an XA-based transaction coordinator, e.g. Spring’s JtaTransactionManager). If not specified, native acknowledging will be used (see "acknowledge" attribute).

concurrency

The number of concurrent sessions/consumers to start for each listener. Can either be a simple number indicating the maximum number (e.g. "5") or a range indicating the lower as well as the upper limit (e.g. "3-5"). Note that a specified minimum is just a hint and might be ignored at runtime. Default is 1; keep concurrency limited to 1 in case of a topic listener or if queue ordering is important; consider raising it for general queues.

prefetch

The maximum number of messages to load into a single session. Note that raising this number might lead to starvation of concurrent consumers!

receive-timeout

The timeout to use for receive calls (in milliseconds). The default is 1000 ms (1 sec); -1 indicates no timeout at all.

back-off

Specify the BackOff instance to use to compute the interval between recovery attempts. If the BackOffExecution implementation returns BackOffExecution#STOP, the listener container will not further attempt to recover. The recovery-interval value is ignored when this property is set. The default is a FixedBackOff with an interval of 5000 ms, that is 5 seconds.

recovery-interval

Specify the interval between recovery attempts, in milliseconds. Convenience way to create a FixedBackOff with the specified interval. For more recovery options, consider specifying a BackOff instance instead. The default is 5000 ms, that is 5 seconds.

phase

The lifecycle phase within which this container should start and stop. The lower the value the earlier this container will start and the later it will stop. The default is Integer.MAX\_VALUE meaning the container will start as late as possible and stop as soon as possible.

Configuring a JCA-based listener container with the "jms" schema support is very similar.

<jms:jca-listener-container resource-adapter="myResourceAdapter"

destination-resolver="myDestinationResolver"

transaction-manager="myTransactionManager"

concurrency="10">

<jms:listener destination="queue.orders" ref="myMessageListener"/>

</jms:jca-listener-container>

The available configuration options for the JCA variant are described in the following table:

Table 5. Attributes of the JMS <jca-listener-container/> element

Attribute Description

factory-id

Exposes the settings defined by this element as a JmsListenerContainerFactory with the specified id so that they can be reused with other endpoints.

resource-adapter

A reference to the JCA ResourceAdapter bean (the default bean name is 'resourceAdapter').

activation-spec-factory

A reference to the JmsActivationSpecFactory. The default is to autodetect the JMS provider and its ActivationSpec class (see DefaultJmsActivationSpecFactory)

destination-resolver

A reference to the DestinationResolver strategy for resolving JMS Destinations.

message-converter

A reference to the MessageConverter strategy for converting JMS Messages to listener method arguments. Default is a SimpleMessageConverter.

destination-type

The JMS destination type for this listener: queue, topic, durableTopic, sharedTopic or sharedDurableTopic. This enables potentially the pubSubDomain, subscriptionDurable and subscriptionShared properties of the container. The default is queue (i.e. disabling those 3 properties).

response-destination-type

The JMS destination type for responses: "queue", "topic". Default is the value of the "destination-type" attribute.

client-id

The JMS client id for this listener container. Needs to be specified when using durable subscriptions.

acknowledge

The native JMS acknowledge mode: auto, client, dups-ok or transacted. A value of transacted activates a locally transacted Session. As an alternative, specify the transaction-manager attribute described below. Default is auto.

transaction-manager

A reference to a Spring JtaTransactionManager or a javax.transaction.TransactionManager for kicking off an XA transaction for each incoming message. If not specified, native acknowledging will be used (see the "acknowledge" attribute).

concurrency

The number of concurrent sessions/consumers to start for each listener. Can either be a simple number indicating the maximum number (e.g. "5") or a range indicating the lower as well as the upper limit (e.g. "3-5"). Note that a specified minimum is just a hint and will typically be ignored at runtime when using a JCA listener container. Default is 1.

prefetch

The maximum number of messages to load into a single session. Note that raising this number might lead to starvation of concurrent consumers!

4. JMX

4.1. Introduction

The JMX support in Spring provides you with the features to easily and transparently integrate your Spring application into a JMX infrastructure.

JMX?

This chapter is not an introduction to JMX…​ it doesn’t try to explain the motivations of why one might want to use JMX (or indeed what the letters JMX actually stand for). If you are new to JMX, check out Further resources at the end of this chapter.

Specifically, Spring’s JMX support provides four core features:

The automatic registration of any Spring bean as a JMX MBean

A flexible mechanism for controlling the management interface of your beans

The declarative exposure of MBeans over remote, JSR-160 connectors

The simple proxying of both local and remote MBean resources

These features are designed to work without coupling your application components to either Spring or JMX interfaces and classes. Indeed, for the most part your application classes need not be aware of either Spring or JMX in order to take advantage of the Spring JMX features.

4.2. Exporting your beans to JMX

The core class in Spring’s JMX framework is the MBeanExporter. This class is responsible for taking your Spring beans and registering them with a JMX MBeanServer. For example, consider the following class:

package org.springframework.jmx;

public class JmxTestBean implements IJmxTestBean {

private String name;

private int age;

private boolean isSuperman;

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public void setName(String name) {

this.name = name;

}

public String getName() {

return name;

}

public int add(int x, int y) {

return x + y;

}

public void dontExposeMe() {

throw new RuntimeException();

}

}

To expose the properties and methods of this bean as attributes and operations of an MBean you simply configure an instance of the MBeanExporter class in your configuration file and pass in the bean as shown below:

<beans>

<!-- this bean must not be lazily initialized if the exporting is to happen -->

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter" lazy-init="false">

<property name="beans">

<map>

<entry key="bean:name=testBean1" value-ref="testBean"/>

</map>

</property>

</bean>

<bean id="testBean" class="org.springframework.jmx.JmxTestBean">

<property name="name" value="TEST"/>

<property name="age" value="100"/>

</bean>

</beans>

The pertinent bean definition from the above configuration snippet is the exporter bean. The beans property tells the MBeanExporter exactly which of your beans must be exported to the JMX MBeanServer. In the default configuration, the key of each entry in the beans Map is used as the ObjectName for the bean referenced by the corresponding entry value. This behavior can be changed as described in Controlling the ObjectNames for your beans.

With this configuration the testBean bean is exposed as an MBean under the ObjectName bean:name=testBean1. By default, all public properties of the bean are exposed as attributes and all public methods (bar those inherited from the Object class) are exposed as operations.

MBeanExporter is a Lifecycle bean (see Startup and shutdown callbacks) and MBeans are exported as late as possible during the application lifecycle by default. It is possible to configure the phase at which the export happens or disable automatic registration by setting the autoStartup flag.

4.2.1. Creating an MBeanServer

The above configuration assumes that the application is running in an environment that has one (and only one) MBeanServer already running. In this case, Spring will attempt to locate the running MBeanServer and register your beans with that server (if any). This behavior is useful when your application is running inside a container such as Tomcat or IBM WebSphere that has its own MBeanServer.

However, this approach is of no use in a standalone environment, or when running inside a container that does not provide an MBeanServer. To address this you can create an MBeanServer instance declaratively by adding an instance of the org.springframework.jmx.support.MBeanServerFactoryBean class to your configuration. You can also ensure that a specific MBeanServer is used by setting the value of the MBeanExporter's server property to the MBeanServer value returned by an MBeanServerFactoryBean; for example:

<beans>

<bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean"/>

<!--

this bean needs to be eagerly pre-instantiated in order for the exporting to occur;

this means that it must not be marked as lazily initialized

-->

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="beans">

<map>

<entry key="bean:name=testBean1" value-ref="testBean"/>

</map>

</property>

<property name="server" ref="mbeanServer"/>

</bean>

<bean id="testBean" class="org.springframework.jmx.JmxTestBean">

<property name="name" value="TEST"/>

<property name="age" value="100"/>

</bean>

</beans>

Here an instance of MBeanServer is created by the MBeanServerFactoryBean and is supplied to the MBeanExporter via the server property. When you supply your own MBeanServer instance, the MBeanExporter will not attempt to locate a running MBeanServer and will use the supplied MBeanServer instance. For this to work correctly, you must (of course) have a JMX implementation on your classpath.

4.2.2. Reusing an existing MBeanServer

If no server is specified, the MBeanExporter tries to automatically detect a running MBeanServer. This works in most environment where only one MBeanServer instance is used, however when multiple instances exist, the exporter might pick the wrong server. In such cases, one should use the MBeanServer agentId to indicate which instance to be used:

<beans>

<bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">

<!-- indicate to first look for a server -->

<property name="locateExistingServerIfPossible" value="true"/>

<!-- search for the MBeanServer instance with the given agentId -->

<property name="agentId" value="MBeanServer\_instance\_agentId>"/>

</bean>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="server" ref="mbeanServer"/>

...

</bean>

</beans>

For platforms/cases where the existing MBeanServer has a dynamic (or unknown) agentId which is retrieved through lookup methods, one should use factory-method:

<beans>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="server">

<!-- Custom MBeanServerLocator -->

<bean class="platform.package.MBeanServerLocator" factory-method="locateMBeanServer"/>

</property>

</bean>

<!-- other beans here -->

</beans>

4.2.3. Lazy-initialized MBeans

If you configure a bean with the MBeanExporter that is also configured for lazy initialization, then the MBeanExporter will not break this contract and will avoid instantiating the bean. Instead, it will register a proxy with the MBeanServer and will defer obtaining the bean from the container until the first invocation on the proxy occurs.

4.2.4. Automatic registration of MBeans

Any beans that are exported through the MBeanExporter and are already valid MBeans are registered as-is with the MBeanServer without further intervention from Spring. MBeans can be automatically detected by the MBeanExporter by setting the autodetect property to true:

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="autodetect" value="true"/>

</bean>

<bean name="spring:mbean=true" class="org.springframework.jmx.export.TestDynamicMBean"/>

Here, the bean called spring:mbean=true is already a valid JMX MBean and will be automatically registered by Spring. By default, beans that are autodetected for JMX registration have their bean name used as the ObjectName. This behavior can be overridden as detailed in Controlling the ObjectNames for your beans.

4.2.5. Controlling the registration behavior

Consider the scenario where a Spring MBeanExporter attempts to register an MBean with an MBeanServer using the ObjectName 'bean:name=testBean1'. If an MBean instance has already been registered under that same ObjectName, the default behavior is to fail (and throw an InstanceAlreadyExistsException).

It is possible to control the behavior of exactly what happens when an MBean is registered with an MBeanServer. Spring’s JMX support allows for three different registration behaviors to control the registration behavior when the registration process finds that an MBean has already been registered under the same ObjectName; these registration behaviors are summarized on the following table:

Table 6. Registration Behaviors

Registration behavior Explanation

REGISTRATION\_FAIL\_ON\_EXISTING

This is the default registration behavior. If an MBean instance has already been registered under the same ObjectName, the MBean that is being registered will not be registered and an InstanceAlreadyExistsException will be thrown. The existing MBean is unaffected.

REGISTRATION\_IGNORE\_EXISTING

If an MBean instance has already been registered under the same ObjectName, the MBean that is being registered will not be registered. The existing MBean is unaffected, and no Exception will be thrown. This is useful in settings where multiple applications want to share a common MBean in a shared MBeanServer.

REGISTRATION\_REPLACE\_EXISTING

If an MBean instance has already been registered under the same ObjectName, the existing MBean that was previously registered will be unregistered and the new MBean will be registered in its place (the new MBean effectively replaces the previous instance).

The above values are defined as constants on the MBeanRegistrationSupport class (the MBeanExporter class derives from this superclass). If you want to change the default registration behavior, you simply need to set the value of the registrationBehaviorName property on your MBeanExporter definition to one of those values.

The following example illustrates how to effect a change from the default registration behavior to the REGISTRATION\_REPLACE\_EXISTING behavior:

<beans>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="beans">

<map>

<entry key="bean:name=testBean1" value-ref="testBean"/>

</map>

</property>

<property name="registrationBehaviorName" value="REGISTRATION\_REPLACE\_EXISTING"/>

</bean>

<bean id="testBean" class="org.springframework.jmx.JmxTestBean">

<property name="name" value="TEST"/>

<property name="age" value="100"/>

</bean>

</beans>

4.3. Controlling the management interface of your beans

In the previous example, you had little control over the management interface of your bean; all of the public properties and methods of each exported bean was exposed as JMX attributes and operations respectively. To exercise finer-grained control over exactly which properties and methods of your exported beans are actually exposed as JMX attributes and operations, Spring JMX provides a comprehensive and extensible mechanism for controlling the management interfaces of your beans.

4.3.1. MBeanInfoAssembler interface

Behind the scenes, the MBeanExporter delegates to an implementation of the org.springframework.jmx.export.assembler.MBeanInfoAssembler interface which is responsible for defining the management interface of each bean that is being exposed. The default implementation, org.springframework.jmx.export.assembler.SimpleReflectiveMBeanInfoAssembler, simply defines a management interface that exposes all public properties and methods (as you saw in the previous examples). Spring provides two additional implementations of the MBeanInfoAssembler interface that allow you to control the generated management interface using either source-level metadata or any arbitrary interface.

4.3.2. Using source-level metadata: Java annotations

Using the MetadataMBeanInfoAssembler you can define the management interfaces for your beans using source level metadata. The reading of metadata is encapsulated by the org.springframework.jmx.export.metadata.JmxAttributeSource interface. Spring JMX provides a default implementation which uses Java annotations, namely org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource. The MetadataMBeanInfoAssembler must be configured with an implementation instance of the JmxAttributeSource interface for it to function correctly (there is no default).

To mark a bean for export to JMX, you should annotate the bean class with the ManagedResource annotation. Each method you wish to expose as an operation must be marked with the ManagedOperation annotation and each property you wish to expose must be marked with the ManagedAttribute annotation. When marking properties you can omit either the annotation of the getter or the setter to create a write-only or read-only attribute respectively.

A ManagedResource annotated bean must be public as well as the methods exposing an operation or an attribute.

The example below shows the annotated version of the JmxTestBean class that you saw earlier:

package org.springframework.jmx;

import org.springframework.jmx.export.annotation.ManagedResource;

import org.springframework.jmx.export.annotation.ManagedOperation;

import org.springframework.jmx.export.annotation.ManagedAttribute;

@ManagedResource(

objectName="bean:name=testBean4",

description="My Managed Bean",

log=true,

logFile="jmx.log",

currencyTimeLimit=15,

persistPolicy="OnUpdate",

persistPeriod=200,

persistLocation="foo",

persistName="bar")

public class AnnotationTestBean implements IJmxTestBean {

private String name;

private int age;

@ManagedAttribute(description="The Age Attribute", currencyTimeLimit=15)

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

@ManagedAttribute(description="The Name Attribute",

currencyTimeLimit=20,

defaultValue="bar",

persistPolicy="OnUpdate")

public void setName(String name) {

this.name = name;

}

@ManagedAttribute(defaultValue="foo", persistPeriod=300)

public String getName() {

return name;

}

@ManagedOperation(description="Add two numbers")

@ManagedOperationParameters({

@ManagedOperationParameter(name = "x", description = "The first number"),

@ManagedOperationParameter(name = "y", description = "The second number")})

public int add(int x, int y) {

return x + y;

}

public void dontExposeMe() {

throw new RuntimeException();

}

}

Here you can see that the JmxTestBean class is marked with the ManagedResource annotation and that this ManagedResource annotation is configured with a set of properties. These properties can be used to configure various aspects of the MBean that is generated by the MBeanExporter, and are explained in greater detail later in section entitled Source-level metadata types.

You will also notice that both the age and name properties are annotated with the ManagedAttribute annotation, but in the case of the age property, only the getter is marked. This will cause both of these properties to be included in the management interface as attributes, but the age attribute will be read-only.

Finally, you will notice that the add(int, int) method is marked with the ManagedOperation attribute whereas the dontExposeMe() method is not. This will cause the management interface to contain only one operation, add(int, int), when using the MetadataMBeanInfoAssembler.

The configuration below shows how you configure the MBeanExporter to use the MetadataMBeanInfoAssembler:

<beans>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="assembler" ref="assembler"/>

<property name="namingStrategy" ref="namingStrategy"/>

<property name="autodetect" value="true"/>

</bean>

<bean id="jmxAttributeSource"

class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

<!-- will create management interface using annotation metadata -->

<bean id="assembler"

class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">

<property name="attributeSource" ref="jmxAttributeSource"/>

</bean>

<!-- will pick up the ObjectName from the annotation -->

<bean id="namingStrategy"

class="org.springframework.jmx.export.naming.MetadataNamingStrategy">

<property name="attributeSource" ref="jmxAttributeSource"/>

</bean>

<bean id="testBean" class="org.springframework.jmx.AnnotationTestBean">

<property name="name" value="TEST"/>

<property name="age" value="100"/>

</bean>

</beans>

Here you can see that an MetadataMBeanInfoAssembler bean has been configured with an instance of the AnnotationJmxAttributeSource class and passed to the MBeanExporter through the assembler property. This is all that is required to take advantage of metadata-driven management interfaces for your Spring-exposed MBeans.

4.3.3. Source-level metadata types

The following source level metadata types are available for use in Spring JMX:

Table 7. Source-level metadata types

Purpose Annotation Annotation Type

Mark all instances of a Class as JMX managed resources

@ManagedResource

Class

Mark a method as a JMX operation

@ManagedOperation

Method

Mark a getter or setter as one half of a JMX attribute

@ManagedAttribute

Method (only getters and setters)

Define descriptions for operation parameters

@ManagedOperationParameter and @ManagedOperationParameters

Method

The following configuration parameters are available for use on these source-level metadata types:

Table 8. Source-level metadata parameters

Parameter Description Applies to

ObjectName

Used by MetadataNamingStrategy to determine the ObjectName of a managed resource

ManagedResource

description

Sets the friendly description of the resource, attribute or operation

ManagedResource, ManagedAttribute, ManagedOperation, ManagedOperationParameter

currencyTimeLimit

Sets the value of the currencyTimeLimit descriptor field

ManagedResource, ManagedAttribute

defaultValue

Sets the value of the defaultValue descriptor field

ManagedAttribute

log

Sets the value of the log descriptor field

ManagedResource

logFile

Sets the value of the logFile descriptor field

ManagedResource

persistPolicy

Sets the value of the persistPolicy descriptor field

ManagedResource

persistPeriod

Sets the value of the persistPeriod descriptor field

ManagedResource

persistLocation

Sets the value of the persistLocation descriptor field

ManagedResource

persistName

Sets the value of the persistName descriptor field

ManagedResource

name

Sets the display name of an operation parameter

ManagedOperationParameter

index

Sets the index of an operation parameter

ManagedOperationParameter

4.3.4. AutodetectCapableMBeanInfoAssembler interface

To simplify configuration even further, Spring introduces the AutodetectCapableMBeanInfoAssembler interface which extends the MBeanInfoAssembler interface to add support for autodetection of MBean resources. If you configure the MBeanExporter with an instance of AutodetectCapableMBeanInfoAssembler then it is allowed to "vote" on the inclusion of beans for exposure to JMX.

Out of the box, the only implementation of the AutodetectCapableMBeanInfo interface is the MetadataMBeanInfoAssembler which will vote to include any bean which is marked with the ManagedResource attribute. The default approach in this case is to use the bean name as the ObjectName which results in a configuration like this:

<beans>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<!-- notice how no 'beans' are explicitly configured here -->

<property name="autodetect" value="true"/>

<property name="assembler" ref="assembler"/>

</bean>

<bean id="testBean" class="org.springframework.jmx.JmxTestBean">

<property name="name" value="TEST"/>

<property name="age" value="100"/>

</bean>

<bean id="assembler" class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">

<property name="attributeSource">

<bean class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

</property>

</bean>

</beans>

Notice that in this configuration no beans are passed to the MBeanExporter; however, the JmxTestBean will still be registered since it is marked with the ManagedResource attribute and the MetadataMBeanInfoAssembler detects this and votes to include it. The only problem with this approach is that the name of the JmxTestBean now has business meaning. You can address this issue by changing the default behavior for ObjectName creation as defined in Controlling the ObjectNames for your beans.

4.3.5. Defining management interfaces using Java interfaces

In addition to the MetadataMBeanInfoAssembler, Spring also includes the InterfaceBasedMBeanInfoAssembler which allows you to constrain the methods and properties that are exposed based on the set of methods defined in a collection of interfaces.

Although the standard mechanism for exposing MBeans is to use interfaces and a simple naming scheme, the InterfaceBasedMBeanInfoAssembler extends this functionality by removing the need for naming conventions, allowing you to use more than one interface and removing the need for your beans to implement the MBean interfaces.

Consider this interface that is used to define a management interface for the JmxTestBean class that you saw earlier:

public interface IJmxTestBean {

public int add(int x, int y);

public long myOperation();

public int getAge();

public void setAge(int age);

public void setName(String name);

public String getName();

}

This interface defines the methods and properties that will be exposed as operations and attributes on the JMX MBean. The code below shows how to configure Spring JMX to use this interface as the definition for the management interface:

<beans>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="beans">

<map>

<entry key="bean:name=testBean5" value-ref="testBean"/>

</map>

</property>

<property name="assembler">

<bean class="org.springframework.jmx.export.assembler.InterfaceBasedMBeanInfoAssembler">

<property name="managedInterfaces">

<value>org.springframework.jmx.IJmxTestBean</value>

</property>

</bean>

</property>

</bean>

<bean id="testBean" class="org.springframework.jmx.JmxTestBean">

<property name="name" value="TEST"/>

<property name="age" value="100"/>

</bean>

</beans>

Here you can see that the InterfaceBasedMBeanInfoAssembler is configured to use the IJmxTestBean interface when constructing the management interface for any bean. It is important to understand that beans processed by the InterfaceBasedMBeanInfoAssembler are not required to implement the interface used to generate the JMX management interface.

In the case above, the IJmxTestBean interface is used to construct all management interfaces for all beans. In many cases this is not the desired behavior and you may want to use different interfaces for different beans. In this case, you can pass InterfaceBasedMBeanInfoAssembler a Properties instance via the interfaceMappings property, where the key of each entry is the bean name and the value of each entry is a comma-separated list of interface names to use for that bean.

If no management interface is specified through either the managedInterfaces or interfaceMappings properties, then the InterfaceBasedMBeanInfoAssembler will reflect on the bean and use all of the interfaces implemented by that bean to create the management interface.

4.3.6. Using MethodNameBasedMBeanInfoAssembler

The MethodNameBasedMBeanInfoAssembler allows you to specify a list of method names that will be exposed to JMX as attributes and operations. The code below shows a sample configuration for this:

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="beans">

<map>

<entry key="bean:name=testBean5" value-ref="testBean"/>

</map>

</property>

<property name="assembler">

<bean class="org.springframework.jmx.export.assembler.MethodNameBasedMBeanInfoAssembler">

<property name="managedMethods">

<value>add,myOperation,getName,setName,getAge</value>

</property>

</bean>

</property>

</bean>

Here you can see that the methods add and myOperation will be exposed as JMX operations and getName(), setName(String) and getAge() will be exposed as the appropriate half of a JMX attribute. In the code above, the method mappings apply to beans that are exposed to JMX. To control method exposure on a bean-by-bean basis, use the methodMappings property of MethodNameMBeanInfoAssembler to map bean names to lists of method names.

4.4. Controlling the ObjectNames for your beans

Behind the scenes, the MBeanExporter delegates to an implementation of the ObjectNamingStrategy to obtain ObjectNames for each of the beans it is registering. The default implementation, KeyNamingStrategy, will, by default, use the key of the beans Map as the ObjectName. In addition, the KeyNamingStrategy can map the key of the beans Map to an entry in a Properties file (or files) to resolve the ObjectName. In addition to the KeyNamingStrategy, Spring provides two additional ObjectNamingStrategy implementations: the IdentityNamingStrategy that builds an ObjectName based on the JVM identity of the bean and the MetadataNamingStrategy that uses source level metadata to obtain the ObjectName.

4.4.1. Reading ObjectNames from Properties

You can configure your own KeyNamingStrategy instance and configure it to read ObjectNames from a Properties instance rather than use bean key. The KeyNamingStrategy will attempt to locate an entry in the Properties with a key corresponding to the bean key. If no entry is found or if the Properties instance is null then the bean key itself is used.

The code below shows a sample configuration for the KeyNamingStrategy:

<beans>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="beans">

<map>

<entry key="testBean" value-ref="testBean"/>

</map>

</property>

<property name="namingStrategy" ref="namingStrategy"/>

</bean>

<bean id="testBean" class="org.springframework.jmx.JmxTestBean">

<property name="name" value="TEST"/>

<property name="age" value="100"/>

</bean>

<bean id="namingStrategy" class="org.springframework.jmx.export.naming.KeyNamingStrategy">

<property name="mappings">

<props>

<prop key="testBean">bean:name=testBean1</prop>

</props>

</property>

<property name="mappingLocations">

<value>names1.properties,names2.properties</value>

</property>

</bean>

</beans>

Here an instance of KeyNamingStrategy is configured with a Properties instance that is merged from the Properties instance defined by the mapping property and the properties files located in the paths defined by the mappings property. In this configuration, the testBean bean will be given the ObjectName bean:name=testBean1 since this is the entry in the Properties instance that has a key corresponding to the bean key.

If no entry in the Properties instance can be found then the bean key name is used as the ObjectName.

4.4.2. Using the MetadataNamingStrategy

The MetadataNamingStrategy uses the objectName property of the ManagedResource attribute on each bean to create the ObjectName. The code below shows the configuration for the MetadataNamingStrategy:

<beans>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="beans">

<map>

<entry key="testBean" value-ref="testBean"/>

</map>

</property>

<property name="namingStrategy" ref="namingStrategy"/>

</bean>

<bean id="testBean" class="org.springframework.jmx.JmxTestBean">

<property name="name" value="TEST"/>

<property name="age" value="100"/>

</bean>

<bean id="namingStrategy" class="org.springframework.jmx.export.naming.MetadataNamingStrategy">

<property name="attributeSource" ref="attributeSource"/>

</bean>

<bean id="attributeSource"

class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

</beans>

If no objectName has been provided for the ManagedResource attribute, then an ObjectName will be created with the following format:[fully-qualified-package-name]:type=[short-classname],name=[bean-name]. For example, the generated ObjectName for the following bean would be: com.foo:type=MyClass,name=myBean.

<bean id="myBean" class="com.foo.MyClass"/>

4.4.3. Configuring annotation based MBean export

If you prefer using the annotation based approach to define your management interfaces, then a convenience subclass of MBeanExporter is available: AnnotationMBeanExporter. When defining an instance of this subclass, the namingStrategy, assembler, and attributeSource configuration is no longer needed, since it will always use standard Java annotation-based metadata (autodetection is always enabled as well). In fact, rather than defining an MBeanExporter bean, an even simpler syntax is supported by the @EnableMBeanExport @Configuration annotation.

@Configuration

@EnableMBeanExport

public class AppConfig {

}

If you prefer XML based configuration the 'context:mbean-export' element serves the same purpose.

<context:mbean-export/>

You can provide a reference to a particular MBean server if necessary, and the defaultDomain attribute (a property of AnnotationMBeanExporter) accepts an alternate value for the generated MBean `ObjectNames’ domains. This would be used in place of the fully qualified package name as described in the previous section on MetadataNamingStrategy.

@EnableMBeanExport(server="myMBeanServer", defaultDomain="myDomain")

@Configuration

ContextConfiguration {

}

<context:mbean-export server="myMBeanServer" default-domain="myDomain"/>

Do not use interface-based AOP proxies in combination with autodetection of JMX annotations in your bean classes. Interface-based proxies 'hide' the target class, which also hides the JMX managed resource annotations. Hence, use target-class proxies in that case: through setting the 'proxy-target-class' flag on <aop:config/>, <tx:annotation-driven/>, etc. Otherwise, your JMX beans might be silently ignored at startup…​

4.5. JSR-160 Connectors

For remote access, Spring JMX module offers two FactoryBean implementations inside the org.springframework.jmx.support package for creating both server- and client-side connectors.

4.5.1. Server-side connectors

To have Spring JMX create, start and expose a JSR-160 JMXConnectorServer use the following configuration:

<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean"/>

By default ConnectorServerFactoryBean creates a JMXConnectorServer bound to "service:jmx:jmxmp://localhost:9875". The serverConnector bean thus exposes the local MBeanServer to clients through the JMXMP protocol on localhost, port 9875. Note that the JMXMP protocol is marked as optional by the JSR 160 specification: currently, the main open-source JMX implementation, MX4J, and the one provided with the JDK do not support JMXMP.

To specify another URL and register the JMXConnectorServer itself with the MBeanServer use the serviceUrl and ObjectName properties respectively:

<bean id="serverConnector"

class="org.springframework.jmx.support.ConnectorServerFactoryBean">

<property name="objectName" value="connector:name=rmi"/>

<property name="serviceUrl"

value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/myconnector"/>

</bean>

If the ObjectName property is set Spring will automatically register your connector with the MBeanServer under that ObjectName. The example below shows the full set of parameters which you can pass to the ConnectorServerFactoryBean when creating a JMXConnector:

<bean id="serverConnector"

class="org.springframework.jmx.support.ConnectorServerFactoryBean">

<property name="objectName" value="connector:name=iiop"/>

<property name="serviceUrl"

value="service:jmx:iiop://localhost/jndi/iiop://localhost:900/myconnector"/>

<property name="threaded" value="true"/>

<property name="daemon" value="true"/>

<property name="environment">

<map>

<entry key="someKey" value="someValue"/>

</map>

</property>

</bean>

Note that when using a RMI-based connector you need the lookup service (tnameserv or rmiregistry) to be started in order for the name registration to complete. If you are using Spring to export remote services for you via RMI, then Spring will already have constructed an RMI registry. If not, you can easily start a registry using the following snippet of configuration:

<bean id="registry" class="org.springframework.remoting.rmi.RmiRegistryFactoryBean">

<property name="port" value="1099"/>

</bean>

4.5.2. Client-side connectors

To create an MBeanServerConnection to a remote JSR-160 enabled MBeanServer use the MBeanServerConnectionFactoryBean as shown below:

<bean id="clientConnector" class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">

<property name="serviceUrl" value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jmxrmi"/>

</bean>

4.5.3. JMX over Hessian or SOAP

JSR-160 permits extensions to the way in which communication is done between the client and the server. The examples above are using the mandatory RMI-based implementation required by the JSR-160 specification (IIOP and JRMP) and the (optional) JMXMP. By using other providers or JMX implementations (such as MX4J) you can take advantage of protocols like SOAP or Hessian over simple HTTP or SSL and others:

<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean">

<property name="objectName" value="connector:name=burlap"/>

<property name="serviceUrl" value="service:jmx:burlap://localhost:9874"/>

</bean>

In the case of the above example, MX4J 3.0.0 was used; see the official MX4J documentation for more information.

4.6. Accessing MBeans via proxies

Spring JMX allows you to create proxies that re-route calls to MBeans registered in a local or remote MBeanServer. These proxies provide you with a standard Java interface through which you can interact with your MBeans. The code below shows how to configure a proxy for an MBean running in a local MBeanServer:

<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">

<property name="objectName" value="bean:name=testBean"/>

<property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>

</bean>

Here you can see that a proxy is created for the MBean registered under the ObjectName: bean:name=testBean. The set of interfaces that the proxy will implement is controlled by the proxyInterfaces property and the rules for mapping methods and properties on these interfaces to operations and attributes on the MBean are the same rules used by the InterfaceBasedMBeanInfoAssembler.

The MBeanProxyFactoryBean can create a proxy to any MBean that is accessible via an MBeanServerConnection. By default, the local MBeanServer is located and used, but you can override this and provide an MBeanServerConnection pointing to a remote MBeanServer to cater for proxies pointing to remote MBeans:

<bean id="clientConnector"

class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">

<property name="serviceUrl" value="service:jmx:rmi://remotehost:9875"/>

</bean>

<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">

<property name="objectName" value="bean:name=testBean"/>

<property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>

<property name="server" ref="clientConnector"/>

</bean>

Here you can see that we create an MBeanServerConnection pointing to a remote machine using the MBeanServerConnectionFactoryBean. This MBeanServerConnection is then passed to the MBeanProxyFactoryBean via the server property. The proxy that is created will forward all invocations to the MBeanServer via this MBeanServerConnection.

4.7. Notifications

Spring’s JMX offering includes comprehensive support for JMX notifications.

4.7.1. Registering listeners for notifications

Spring’s JMX support makes it very easy to register any number of NotificationListeners with any number of MBeans (this includes MBeans exported by Spring’s MBeanExporter and MBeans registered via some other mechanism). By way of an example, consider the scenario where one would like to be informed (via a Notification) each and every time an attribute of a target MBean changes.

package com.example;

import javax.management.AttributeChangeNotification;

import javax.management.Notification;

import javax.management.NotificationFilter;

import javax.management.NotificationListener;

public class ConsoleLoggingNotificationListener

implements NotificationListener, NotificationFilter {

public void handleNotification(Notification notification, Object handback) {

System.out.println(notification);

System.out.println(handback);

}

public boolean isNotificationEnabled(Notification notification) {

return AttributeChangeNotification.class.isAssignableFrom(notification.getClass());

}

}

<beans>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="beans">

<map>

<entry key="bean:name=testBean1" value-ref="testBean"/>

</map>

</property>

<property name="notificationListenerMappings">

<map>

<entry key="bean:name=testBean1">

<bean class="com.example.ConsoleLoggingNotificationListener"/>

</entry>

</map>

</property>

</bean>

<bean id="testBean" class="org.springframework.jmx.JmxTestBean">

<property name="name" value="TEST"/>

<property name="age" value="100"/>

</bean>

</beans>

With the above configuration in place, every time a JMX Notification is broadcast from the target MBean ( bean:name=testBean1), the ConsoleLoggingNotificationListener bean that was registered as a listener via the notificationListenerMappings property will be notified. The ConsoleLoggingNotificationListener bean can then take whatever action it deems appropriate in response to the Notification.

You can also use straight bean names as the link between exported beans and listeners:

<beans>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="beans">

<map>

<entry key="bean:name=testBean1" value-ref="testBean"/>

</map>

</property>

<property name="notificationListenerMappings">

<map>

<entry key="testBean">

<bean class="com.example.ConsoleLoggingNotificationListener"/>

</entry>

</map>

</property>

</bean>

<bean id="testBean" class="org.springframework.jmx.JmxTestBean">

<property name="name" value="TEST"/>

<property name="age" value="100"/>

</bean>

</beans>

If one wants to register a single NotificationListener instance for all of the beans that the enclosing MBeanExporter is exporting, one can use the special wildcard '\*' (sans quotes) as the key for an entry in the notificationListenerMappings property map; for example:

<property name="notificationListenerMappings">

<map>

<entry key="\*">

<bean class="com.example.ConsoleLoggingNotificationListener"/>

</entry>

</map>

</property>

If one needs to do the inverse (that is, register a number of distinct listeners against an MBean), then one has to use the notificationListeners list property instead (and in preference to the notificationListenerMappings property). This time, instead of configuring simply a NotificationListener for a single MBean, one configures NotificationListenerBean instances…​ a NotificationListenerBean encapsulates a NotificationListener and the ObjectName (or ObjectNames) that it is to be registered against in an MBeanServer. The NotificationListenerBean also encapsulates a number of other properties such as a NotificationFilter and an arbitrary handback object that can be used in advanced JMX notification scenarios.

The configuration when using NotificationListenerBean instances is not wildly different to what was presented previously:

<beans>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="beans">

<map>

<entry key="bean:name=testBean1" value-ref="testBean"/>

</map>

</property>

<property name="notificationListeners">

<list>

<bean class="org.springframework.jmx.export.NotificationListenerBean">

<constructor-arg>

<bean class="com.example.ConsoleLoggingNotificationListener"/>

</constructor-arg>

<property name="mappedObjectNames">

<list>

<value>bean:name=testBean1</value>

</list>

</property>

</bean>

</list>

</property>

</bean>

<bean id="testBean" class="org.springframework.jmx.JmxTestBean">

<property name="name" value="TEST"/>

<property name="age" value="100"/>

</bean>

</beans>

The above example is equivalent to the first notification example. Lets assume then that we want to be given a handback object every time a Notification is raised, and that additionally we want to filter out extraneous Notifications by supplying a NotificationFilter. (For a full discussion of just what a handback object is, and indeed what a NotificationFilter is, please do consult that section of the JMX specification (1.2) entitled 'The JMX Notification Model'.)

<beans>

<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">

<property name="beans">

<map>

<entry key="bean:name=testBean1" value-ref="testBean1"/>

<entry key="bean:name=testBean2" value-ref="testBean2"/>

</map>

</property>

<property name="notificationListeners">

<list>

<bean class="org.springframework.jmx.export.NotificationListenerBean">

<constructor-arg ref="customerNotificationListener"/>

<property name="mappedObjectNames">

<list>

<!-- handles notifications from two distinct MBeans -->

<value>bean:name=testBean1</value>

<value>bean:name=testBean2</value>

</list>

</property>

<property name="handback">

<bean class="java.lang.String">

<constructor-arg value="This could be anything..."/>

</bean>

</property>

<property name="notificationFilter" ref="customerNotificationListener"/>

</bean>

</list>

</property>

</bean>

<!-- implements both the NotificationListener and NotificationFilter interfaces -->

<bean id="customerNotificationListener" class="com.example.ConsoleLoggingNotificationListener"/>

<bean id="testBean1" class="org.springframework.jmx.JmxTestBean">

<property name="name" value="TEST"/>

<property name="age" value="100"/>

</bean>

<bean id="testBean2" class="org.springframework.jmx.JmxTestBean">

<property name="name" value="ANOTHER TEST"/>

<property name="age" value="200"/>

</bean>

</beans>

4.7.2. Publishing Notifications

Spring provides support not just for registering to receive Notifications, but also for publishing Notifications.

Please note that this section is really only relevant to Spring managed beans that have been exposed as MBeans via an MBeanExporter; any existing, user-defined MBeans should use the standard JMX APIs for notification publication.

The key interface in Spring’s JMX notification publication support is the NotificationPublisher interface (defined in the org.springframework.jmx.export.notification package). Any bean that is going to be exported as an MBean via an MBeanExporter instance can implement the related NotificationPublisherAware interface to gain access to a NotificationPublisher instance. The NotificationPublisherAware interface simply supplies an instance of a NotificationPublisher to the implementing bean via a simple setter method, which the bean can then use to publish Notifications.

As stated in the javadocs of the NotificationPublisher class, managed beans that are publishing events via the NotificationPublisher mechanism are not responsible for the state management of any notification listeners and the like …​ Spring’s JMX support will take care of handling all the JMX infrastructure issues. All one need do as an application developer is implement the NotificationPublisherAware interface and start publishing events using the supplied NotificationPublisher instance. Note that the NotificationPublisher will be set after the managed bean has been registered with an MBeanServer.

Using a NotificationPublisher instance is quite straightforward…​ one simply creates a JMX Notification instance (or an instance of an appropriate Notification subclass), populates the notification with the data pertinent to the event that is to be published, and one then invokes the sendNotification(Notification) on the NotificationPublisher instance, passing in the Notification.

Find below a simple example…​ in this scenario, exported instances of the JmxTestBean are going to publish a NotificationEvent every time the add(int, int) operation is invoked.

package org.springframework.jmx;

import org.springframework.jmx.export.notification.NotificationPublisherAware;

import org.springframework.jmx.export.notification.NotificationPublisher;

import javax.management.Notification;

public class JmxTestBean implements IJmxTestBean, NotificationPublisherAware {

private String name;

private int age;

private boolean isSuperman;

private NotificationPublisher publisher;

// other getters and setters omitted for clarity

public int add(int x, int y) {

int answer = x + y;

this.publisher.sendNotification(new Notification("add", this, 0));

return answer;

}

public void dontExposeMe() {

throw new RuntimeException();

}

public void setNotificationPublisher(NotificationPublisher notificationPublisher) {

this.publisher = notificationPublisher;

}

}

The NotificationPublisher interface and the machinery to get it all working is one of the nicer features of Spring’s JMX support. It does however come with the price tag of coupling your classes to both Spring and JMX; as always, the advice here is to be pragmatic…​ if you need the functionality offered by the NotificationPublisher and you can accept the coupling to both Spring and JMX, then do so.

4.8. Further resources

This section contains links to further resources about JMX.

The JMX homepage at Oracle

The JMX specification (JSR-000003)

The JMX Remote API specification (JSR-000160)

The MX4J homepage (an Open Source implementation of various JMX specs)

5. JCA CCI

5.1. Introduction

Java EE provides a specification to standardize access to enterprise information systems (EIS): the JCA (Java EE Connector Architecture). This specification is divided into several different parts:

SPI (Service provider interfaces) that the connector provider must implement. These interfaces constitute a resource adapter which can be deployed on a Java EE application server. In such a scenario, the server manages connection pooling, transaction and security (managed mode). The application server is also responsible for managing the configuration, which is held outside the client application. A connector can be used without an application server as well; in this case, the application must configure it directly (non-managed mode).

CCI (Common Client Interface) that an application can use to interact with the connector and thus communicate with an EIS. An API for local transaction demarcation is provided as well.

The aim of the Spring CCI support is to provide classes to access a CCI connector in typical Spring style, leveraging the Spring Framework’s general resource and transaction management facilities.

The client side of connectors doesn’t alway use CCI. Some connectors expose their own APIs, only providing JCA resource adapter to use the system contracts of a Java EE container (connection pooling, global transactions, security). Spring does not offer special support for such connector-specific APIs.

5.2. Configuring CCI

5.2.1. Connector configuration

The base resource to use JCA CCI is the ConnectionFactory interface. The connector used must provide an implementation of this interface.

To use your connector, you can deploy it on your application server and fetch the ConnectionFactory from the server’s JNDI environment (managed mode). The connector must be packaged as a RAR file (resource adapter archive) and contain a ra.xml file to describe its deployment characteristics. The actual name of the resource is specified when you deploy it. To access it within Spring, simply use Spring’s JndiObjectFactoryBean / <jee:jndi-lookup> fetch the factory by its JNDI name.

Another way to use a connector is to embed it in your application (non-managed mode), not using an application server to deploy and configure it. Spring offers the possibility to configure a connector as a bean, through a provided FactoryBean ( LocalConnectionFactoryBean). In this manner, you only need the connector library in the classpath (no RAR file and no ra.xml descriptor needed). The library must be extracted from the connector’s RAR file, if necessary.

Once you have got access to your ConnectionFactory instance, you can inject it into your components. These components can either be coded against the plain CCI API or leverage Spring’s support classes for CCI access (e.g. CciTemplate).

When you use a connector in non-managed mode, you can’t use global transactions because the resource is never enlisted / delisted in the current global transaction of the current thread. The resource is simply not aware of any global Java EE transactions that might be running.

5.2.2. ConnectionFactory configuration in Spring

In order to make connections to the EIS, you need to obtain a ConnectionFactory from the application server if you are in a managed mode, or directly from Spring if you are in a non-managed mode.

In a managed mode, you access a ConnectionFactory from JNDI; its properties will be configured in the application server.

<jee:jndi-lookup id="eciConnectionFactory" jndi-name="eis/cicseci"/>

In non-managed mode, you must configure the ConnectionFactory you want to use in the configuration of Spring as a JavaBean. The LocalConnectionFactoryBean class offers this setup style, passing in the ManagedConnectionFactory implementation of your connector, exposing the application-level CCI ConnectionFactory.

<bean id="eciManagedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">

<property name="serverName" value="TXSERIES"/>

<property name="connectionURL" value="tcp://localhost/"/>

<property name="portNumber" value="2006"/>

</bean>

<bean id="eciConnectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">

<property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>

</bean>

You can’t directly instantiate a specific ConnectionFactory. You need to go through the corresponding implementation of the ManagedConnectionFactory interface for your connector. This interface is part of the JCA SPI specification.

5.2.3. Configuring CCI connections

JCA CCI allow the developer to configure the connections to the EIS using the ConnectionSpec implementation of your connector. In order to configure its properties, you need to wrap the target connection factory with a dedicated adapter, ConnectionSpecConnectionFactoryAdapter. So, the dedicated ConnectionSpec can be configured with the property connectionSpec (as an inner bean).

This property is not mandatory because the CCI ConnectionFactory interface defines two different methods to obtain a CCI connection. Some of the ConnectionSpec properties can often be configured in the application server (in managed mode) or on the corresponding local ManagedConnectionFactory implementation.

public interface ConnectionFactory implements Serializable, Referenceable {

...

Connection getConnection() throws ResourceException;

Connection getConnection(ConnectionSpec connectionSpec) throws ResourceException;

...

}

Spring provides a ConnectionSpecConnectionFactoryAdapter that allows for specifying a ConnectionSpec instance to use for all operations on a given factory. If the adapter’s connectionSpec property is specified, the adapter uses the getConnection variant with the ConnectionSpec argument, otherwise the variant without argument.

<bean id="managedConnectionFactory"

class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">

<property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>

<property name="driverName" value="org.hsqldb.jdbcDriver"/>

</bean>

<bean id="targetConnectionFactory"

class="org.springframework.jca.support.LocalConnectionFactoryBean">

<property name="managedConnectionFactory" ref="managedConnectionFactory"/>

</bean>

<bean id="connectionFactory"

class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">

<property name="targetConnectionFactory" ref="targetConnectionFactory"/>

<property name="connectionSpec">

<bean class="com.sun.connector.cciblackbox.CciConnectionSpec">

<property name="user" value="sa"/>

<property name="password" value=""/>

</bean>

</property>

</bean>

5.2.4. Using a single CCI connection

If you want to use a single CCI connection, Spring provides a further ConnectionFactory adapter to manage this. The SingleConnectionFactory adapter class will open a single connection lazily and close it when this bean is destroyed at application shutdown. This class will expose special Connection proxies that behave accordingly, all sharing the same underlying physical connection.

<bean id="eciManagedConnectionFactory"

class="com.ibm.connector2.cics.ECIManagedConnectionFactory">

<property name="serverName" value="TEST"/>

<property name="connectionURL" value="tcp://localhost/"/>

<property name="portNumber" value="2006"/>

</bean>

<bean id="targetEciConnectionFactory"

class="org.springframework.jca.support.LocalConnectionFactoryBean">

<property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>

</bean>

<bean id="eciConnectionFactory"

class="org.springframework.jca.cci.connection.SingleConnectionFactory">

<property name="targetConnectionFactory" ref="targetEciConnectionFactory"/>

</bean>

This ConnectionFactory adapter cannot directly be configured with a ConnectionSpec. Use an intermediary ConnectionSpecConnectionFactoryAdapter that the SingleConnectionFactory talks to if you require a single connection for a specific ConnectionSpec.

5.3. Using Spring’s CCI access support

5.3.1. Record conversion

One of the aims of the JCA CCI support is to provide convenient facilities for manipulating CCI records. The developer can specify the strategy to create records and extract datas from records, for use with Spring’s CciTemplate. The following interfaces will configure the strategy to use input and output records if you don’t want to work with records directly in your application.

In order to create an input Record, the developer can use a dedicated implementation of the RecordCreator interface.

public interface RecordCreator {

Record createRecord(RecordFactory recordFactory) throws ResourceException, DataAccessException;

}

As you can see, the createRecord(..) method receives a RecordFactory instance as parameter, which corresponds to the RecordFactory of the ConnectionFactory used. This reference can be used to create IndexedRecord or MappedRecord instances. The following sample shows how to use the RecordCreator interface and indexed/mapped records.

public class MyRecordCreator implements RecordCreator {

public Record createRecord(RecordFactory recordFactory) throws ResourceException {

IndexedRecord input = recordFactory.createIndexedRecord("input");

input.add(new Integer(id));

return input;

}

}

An output Record can be used to receive data back from the EIS. Hence, a specific implementation of the RecordExtractor interface can be passed to Spring’s CciTemplate for extracting data from the output Record.

public interface RecordExtractor {

Object extractData(Record record) throws ResourceException, SQLException, DataAccessException;

}

The following sample shows how to use the RecordExtractor interface.

public class MyRecordExtractor implements RecordExtractor {

public Object extractData(Record record) throws ResourceException {

CommAreaRecord commAreaRecord = (CommAreaRecord) record;

String str = new String(commAreaRecord.toByteArray());

String field1 = string.substring(0,6);

String field2 = string.substring(6,1);

return new OutputObject(Long.parseLong(field1), field2);

}

}

5.3.2. CciTemplate

The CciTemplate is the central class of the core CCI support package ( org.springframework.jca.cci.core). It simplifies the use of CCI since it handles the creation and release of resources. This helps to avoid common errors like forgetting to always close the connection. It cares for the lifecycle of connection and interaction objects, letting application code focus on generating input records from application data and extracting application data from output records.

The JCA CCI specification defines two distinct methods to call operations on an EIS. The CCI Interaction interface provides two execute method signatures:

public interface javax.resource.cci.Interaction {

...

boolean execute(InteractionSpec spec, Record input, Record output) throws ResourceException;

Record execute(InteractionSpec spec, Record input) throws ResourceException;

...

}

Depending on the template method called, CciTemplate will know which execute method to call on the interaction. In any case, a correctly initialized InteractionSpec instance is mandatory.

CciTemplate.execute(..) can be used in two ways:

With direct Record arguments. In this case, you simply need to pass the CCI input record in, and the returned object be the corresponding CCI output record.

With application objects, using record mapping. In this case, you need to provide corresponding RecordCreator and RecordExtractor instances.

With the first approach, the following methods of the template will be used. These methods directly correspond to those on the Interaction interface.

public class CciTemplate implements CciOperations {

public Record execute(InteractionSpec spec, Record inputRecord)

throws DataAccessException { ... }

public void execute(InteractionSpec spec, Record inputRecord, Record outputRecord)

throws DataAccessException { ... }

}

With the second approach, we need to specify the record creation and record extraction strategies as arguments. The interfaces used are those describe in the previous section on record conversion. The corresponding CciTemplate methods are the following:

public class CciTemplate implements CciOperations {

public Record execute(InteractionSpec spec,

RecordCreator inputCreator) throws DataAccessException {

// ...

}

public Object execute(InteractionSpec spec, Record inputRecord,

RecordExtractor outputExtractor) throws DataAccessException {

// ...

}

public Object execute(InteractionSpec spec, RecordCreator creator,

RecordExtractor extractor) throws DataAccessException {

// ...

}

}

Unless the outputRecordCreator property is set on the template (see the following section), every method will call the corresponding execute method of the CCI Interaction with two parameters: InteractionSpec and input Record, receiving an output Record as return value.

CciTemplate also provides methods to create IndexRecord and MappedRecord outside a RecordCreator implementation, through its createIndexRecord(..) and createMappedRecord(..) methods. This can be used within DAO implementations to create Record instances to pass into corresponding CciTemplate.execute(..) methods.

public class CciTemplate implements CciOperations {

public IndexedRecord createIndexedRecord(String name) throws DataAccessException { ... }

public MappedRecord createMappedRecord(String name) throws DataAccessException { ... }

}

5.3.3. DAO support

Spring’s CCI support provides a abstract class for DAOs, supporting injection of a ConnectionFactory or a CciTemplate instances. The name of the class is CciDaoSupport: It provides simple setConnectionFactory and setCciTemplate methods. Internally, this class will create a CciTemplate instance for a passed-in ConnectionFactory, exposing it to concrete data access implementations in subclasses.

public abstract class CciDaoSupport {

public void setConnectionFactory(ConnectionFactory connectionFactory) {

// ...

}

public ConnectionFactory getConnectionFactory() {

// ...

}

public void setCciTemplate(CciTemplate cciTemplate) {

// ...

}

public CciTemplate getCciTemplate() {

// ...

}

}

5.3.4. Automatic output record generation

If the connector used only supports the Interaction.execute(..) method with input and output records as parameters (that is, it requires the desired output record to be passed in instead of returning an appropriate output record), you can set the outputRecordCreator property of the CciTemplate to automatically generate an output record to be filled by the JCA connector when the response is received. This record will be then returned to the caller of the template.

This property simply holds an implementation of the RecordCreator interface, used for that purpose. The RecordCreator interface has already been discussed in Record conversion. The outputRecordCreator property must be directly specified on the CciTemplate. This could be done in the application code like so:

cciTemplate.setOutputRecordCreator(new EciOutputRecordCreator());

Or (recommended) in the Spring configuration, if the CciTemplate is configured as a dedicated bean instance:

<bean id="eciOutputRecordCreator" class="eci.EciOutputRecordCreator"/>

<bean id="cciTemplate" class="org.springframework.jca.cci.core.CciTemplate">

<property name="connectionFactory" ref="eciConnectionFactory"/>

<property name="outputRecordCreator" ref="eciOutputRecordCreator"/>

</bean>

As the CciTemplate class is thread-safe, it will usually be configured as a shared instance.

5.3.5. Summary

The following table summarizes the mechanisms of the CciTemplate class and the corresponding methods called on the CCI Interaction interface:

Table 9. Usage of Interaction execute methods

CciTemplate method signature CciTemplate outputRecordCreator property execute method called on the CCI Interaction

Record execute(InteractionSpec, Record)

not set

Record execute(InteractionSpec, Record)

Record execute(InteractionSpec, Record)

set

boolean execute(InteractionSpec, Record, Record)

void execute(InteractionSpec, Record, Record)

not set

void execute(InteractionSpec, Record, Record)

void execute(InteractionSpec, Record, Record)

set

void execute(InteractionSpec, Record, Record)

Record execute(InteractionSpec, RecordCreator)

not set

Record execute(InteractionSpec, Record)

Record execute(InteractionSpec, RecordCreator)

set

void execute(InteractionSpec, Record, Record)

Record execute(InteractionSpec, Record, RecordExtractor)

not set

Record execute(InteractionSpec, Record)

Record execute(InteractionSpec, Record, RecordExtractor)

set

void execute(InteractionSpec, Record, Record)

Record execute(InteractionSpec, RecordCreator, RecordExtractor)

not set

Record execute(InteractionSpec, Record)

Record execute(InteractionSpec, RecordCreator, RecordExtractor)

set

void execute(InteractionSpec, Record, Record)

5.3.6. Using a CCI Connection and Interaction directly

CciTemplate also offers the possibility to work directly with CCI connections and interactions, in the same manner as JdbcTemplate and JmsTemplate. This is useful when you want to perform multiple operations on a CCI connection or interaction, for example.

The interface ConnectionCallback provides a CCI Connection as argument, in order to perform custom operations on it, plus the CCI ConnectionFactory which the Connection was created with. The latter can be useful for example to get an associated RecordFactory instance and create indexed/mapped records, for example.

public interface ConnectionCallback {

Object doInConnection(Connection connection, ConnectionFactory connectionFactory)

throws ResourceException, SQLException, DataAccessException;

}

The interface InteractionCallback provides the CCI Interaction, in order to perform custom operations on it, plus the corresponding CCI ConnectionFactory.

public interface InteractionCallback {

Object doInInteraction(Interaction interaction, ConnectionFactory connectionFactory)

throws ResourceException, SQLException, DataAccessException;

}

InteractionSpec objects can either be shared across multiple template calls or newly created inside every callback method. This is completely up to the DAO implementation.

5.3.7. Example for CciTemplate usage

In this section, the usage of the CciTemplate will be shown to acces to a CICS with ECI mode, with the IBM CICS ECI connector.

Firstly, some initializations on the CCI InteractionSpec must be done to specify which CICS program to access and how to interact with it.

ECIInteractionSpec interactionSpec = new ECIInteractionSpec();

interactionSpec.setFunctionName("MYPROG");

interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC\_SEND\_RECEIVE);

Then the program can use CCI via Spring’s template and specify mappings between custom objects and CCI Records.

public class MyDaoImpl extends CciDaoSupport implements MyDao {

public OutputObject getData(InputObject input) {

ECIInteractionSpec interactionSpec = ...;

OutputObject output = (ObjectOutput) getCciTemplate().execute(interactionSpec,

new RecordCreator() {

public Record createRecord(RecordFactory recordFactory) throws ResourceException {

return new CommAreaRecord(input.toString().getBytes());

}

},

new RecordExtractor() {

public Object extractData(Record record) throws ResourceException {

CommAreaRecord commAreaRecord = (CommAreaRecord)record;

String str = new String(commAreaRecord.toByteArray());

String field1 = string.substring(0,6);

String field2 = string.substring(6,1);

return new OutputObject(Long.parseLong(field1), field2);

}

});

return output;

}

}

As discussed previously, callbacks can be used to work directly on CCI connections or interactions.

public class MyDaoImpl extends CciDaoSupport implements MyDao {

public OutputObject getData(InputObject input) {

ObjectOutput output = (ObjectOutput) getCciTemplate().execute(

new ConnectionCallback() {

public Object doInConnection(Connection connection,

ConnectionFactory factory) throws ResourceException {

// do something...

}

});

}

return output;

}

}

With a ConnectionCallback, the Connection used will be managed and closed by the CciTemplate, but any interactions created on the connection must be managed by the callback implementation.

For a more specific callback, you can implement an InteractionCallback. The passed-in Interaction will be managed and closed by the CciTemplate in this case.

public class MyDaoImpl extends CciDaoSupport implements MyDao {

public String getData(String input) {

ECIInteractionSpec interactionSpec = ...;

String output = (String) getCciTemplate().execute(interactionSpec,

new InteractionCallback() {

public Object doInInteraction(Interaction interaction,

ConnectionFactory factory) throws ResourceException {

Record input = new CommAreaRecord(inputString.getBytes());

Record output = new CommAreaRecord();

interaction.execute(holder.getInteractionSpec(), input, output);

return new String(output.toByteArray());

}

});

return output;

}

}

For the examples above, the corresponding configuration of the involved Spring beans could look like this in non-managed mode:

<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">

<property name="serverName" value="TXSERIES"/>

<property name="connectionURL" value="local:"/>

<property name="userName" value="CICSUSER"/>

<property name="password" value="CICS"/>

</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">

<property name="managedConnectionFactory" ref="managedConnectionFactory"/>

</bean>

<bean id="component" class="mypackage.MyDaoImpl">

<property name="connectionFactory" ref="connectionFactory"/>

</bean>

In managed mode (that is, in a Java EE environment), the configuration could look as follows:

<jee:jndi-lookup id="connectionFactory" jndi-name="eis/cicseci"/>

<bean id="component" class="MyDaoImpl">

<property name="connectionFactory" ref="connectionFactory"/>

</bean>

5.4. Modeling CCI access as operation objects

The org.springframework.jca.cci.object package contains support classes that allow you to access the EIS in a different style: through reusable operation objects, analogous to Spring’s JDBC operation objects (see JDBC chapter). This will usually encapsulate the CCI API: an application-level input object will be passed to the operation object, so it can construct the input record and then convert the received record data to an application-level output object and return it.

This approach is internally based on the CciTemplate class and the RecordCreator / RecordExtractor interfaces, reusing the machinery of Spring’s core CCI support.

5.4.1. MappingRecordOperation

MappingRecordOperation essentially performs the same work as CciTemplate, but represents a specific, pre-configured operation as an object. It provides two template methods to specify how to convert an input object to a input record, and how to convert an output record to an output object (record mapping):

createInputRecord(..) to specify how to convert an input object to an input Record

extractOutputData(..) to specify how to extract an output object from an output Record

Here are the signatures of these methods:

public abstract class MappingRecordOperation extends EisOperation {

...

protected abstract Record createInputRecord(RecordFactory recordFactory,

Object inputObject) throws ResourceException, DataAccessException {

// ...

}

protected abstract Object extractOutputData(Record outputRecord)

throws ResourceException, SQLException, DataAccessException {

// ...

}

...

}

Thereafter, in order to execute an EIS operation, you need to use a single execute method, passing in an application-level input object and receiving an application-level output object as result:

public abstract class MappingRecordOperation extends EisOperation {

...

public Object execute(Object inputObject) throws DataAccessException {

}

...

}

As you can see, contrary to the CciTemplate class, this execute(..) method does not have an InteractionSpec as argument. Instead, the InteractionSpec is global to the operation. The following constructor must be used to instantiate an operation object with a specific InteractionSpec:

InteractionSpec spec = ...;

MyMappingRecordOperation eisOperation = new MyMappingRecordOperation(getConnectionFactory(), spec);

...

5.4.2. MappingCommAreaOperation

Some connectors use records based on a COMMAREA which represents an array of bytes containing parameters to send to the EIS and data returned by it. Spring provides a special operation class for working directly on COMMAREA rather than on records. The MappingCommAreaOperation class extends the MappingRecordOperation class to provide such special COMMAREA support. It implicitly uses the CommAreaRecord class as input and output record type, and provides two new methods to convert an input object into an input COMMAREA and the output COMMAREA into an output object.

public abstract class MappingCommAreaOperation extends MappingRecordOperation {

...

protected abstract byte[] objectToBytes(Object inObject)

throws IOException, DataAccessException;

protected abstract Object bytesToObject(byte[] bytes)

throws IOException, DataAccessException;

...

}

5.4.3. Automatic output record generation

As every MappingRecordOperation subclass is based on CciTemplate internally, the same way to automatically generate output records as with CciTemplate is available. Every operation object provides a corresponding setOutputRecordCreator(..) method. For further information, see Automatic output record generation.

5.4.4. Summary

The operation object approach uses records in the same manner as the CciTemplate class.

Table 10. Usage of Interaction execute methods

MappingRecordOperation method signature MappingRecordOperation outputRecordCreator property execute method called on the CCI Interaction

Object execute(Object)

not set

Record execute(InteractionSpec, Record)

Object execute(Object)

set

boolean execute(InteractionSpec, Record, Record)

5.4.5. Example for MappingRecordOperation usage

In this section, the usage of the MappingRecordOperation will be shown to access a database with the Blackbox CCI connector.

The original version of this connector is provided by the Java EE SDK (version 1.3), available from Oracle.

Firstly, some initializations on the CCI InteractionSpec must be done to specify which SQL request to execute. In this sample, we directly define the way to convert the parameters of the request to a CCI record and the way to convert the CCI result record to an instance of the Person class.

public class PersonMappingOperation extends MappingRecordOperation {

public PersonMappingOperation(ConnectionFactory connectionFactory) {

setConnectionFactory(connectionFactory);

CciInteractionSpec interactionSpec = new CciConnectionSpec();

interactionSpec.setSql("select \* from person where person\_id=?");

setInteractionSpec(interactionSpec);

}

protected Record createInputRecord(RecordFactory recordFactory,

Object inputObject) throws ResourceException {

Integer id = (Integer) inputObject;

IndexedRecord input = recordFactory.createIndexedRecord("input");

input.add(new Integer(id));

return input;

}

protected Object extractOutputData(Record outputRecord)

throws ResourceException, SQLException {

ResultSet rs = (ResultSet) outputRecord;

Person person = null;

if (rs.next()) {

Person person = new Person();

person.setId(rs.getInt("person\_id"));

person.setLastName(rs.getString("person\_last\_name"));

person.setFirstName(rs.getString("person\_first\_name"));

}

return person;

}

}

Then the application can execute the operation object, with the person identifier as argument. Note that operation object could be set up as shared instance, as it is thread-safe.

public class MyDaoImpl extends CciDaoSupport implements MyDao {

public Person getPerson(int id) {

PersonMappingOperation query = new PersonMappingOperation(getConnectionFactory());

Person person = (Person) query.execute(new Integer(id));

return person;

}

}

The corresponding configuration of Spring beans could look as follows in non-managed mode:

<bean id="managedConnectionFactory"

class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">

<property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>

<property name="driverName" value="org.hsqldb.jdbcDriver"/>

</bean>

<bean id="targetConnectionFactory"

class="org.springframework.jca.support.LocalConnectionFactoryBean">

<property name="managedConnectionFactory" ref="managedConnectionFactory"/>

</bean>

<bean id="connectionFactory"

class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">

<property name="targetConnectionFactory" ref="targetConnectionFactory"/>

<property name="connectionSpec">

<bean class="com.sun.connector.cciblackbox.CciConnectionSpec">

<property name="user" value="sa"/>

<property name="password" value=""/>

</bean>

</property>

</bean>

<bean id="component" class="MyDaoImpl">

<property name="connectionFactory" ref="connectionFactory"/>

</bean>

In managed mode (that is, in a Java EE environment), the configuration could look as follows:

<jee:jndi-lookup id="targetConnectionFactory" jndi-name="eis/blackbox"/>

<bean id="connectionFactory"

class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">

<property name="targetConnectionFactory" ref="targetConnectionFactory"/>

<property name="connectionSpec">

<bean class="com.sun.connector.cciblackbox.CciConnectionSpec">

<property name="user" value="sa"/>

<property name="password" value=""/>

</bean>

</property>

</bean>

<bean id="component" class="MyDaoImpl">

<property name="connectionFactory" ref="connectionFactory"/>

</bean>

5.4.6. Example for MappingCommAreaOperation usage

In this section, the usage of the MappingCommAreaOperation will be shown: accessing a CICS with ECI mode with the IBM CICS ECI connector.

Firstly, the CCI InteractionSpec needs to be initialized to specify which CICS program to access and how to interact with it.

public abstract class EciMappingOperation extends MappingCommAreaOperation {

public EciMappingOperation(ConnectionFactory connectionFactory, String programName) {

setConnectionFactory(connectionFactory);

ECIInteractionSpec interactionSpec = new ECIInteractionSpec(),

interactionSpec.setFunctionName(programName);

interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC\_SEND\_RECEIVE);

interactionSpec.setCommareaLength(30);

setInteractionSpec(interactionSpec);

setOutputRecordCreator(new EciOutputRecordCreator());

}

private static class EciOutputRecordCreator implements RecordCreator {

public Record createRecord(RecordFactory recordFactory) throws ResourceException {

return new CommAreaRecord();

}

}

}

The abstract EciMappingOperation class can then be subclassed to specify mappings between custom objects and Records.

public class MyDaoImpl extends CciDaoSupport implements MyDao {

public OutputObject getData(Integer id) {

EciMappingOperation query = new EciMappingOperation(getConnectionFactory(), "MYPROG") {

protected abstract byte[] objectToBytes(Object inObject) throws IOException {

Integer id = (Integer) inObject;

return String.valueOf(id);

}

protected abstract Object bytesToObject(byte[] bytes) throws IOException;

String str = new String(bytes);

String field1 = str.substring(0,6);

String field2 = str.substring(6,1);

String field3 = str.substring(7,1);

return new OutputObject(field1, field2, field3);

}

});

return (OutputObject) query.execute(new Integer(id));

}

}

The corresponding configuration of Spring beans could look as follows in non-managed mode:

<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">

<property name="serverName" value="TXSERIES"/>

<property name="connectionURL" value="local:"/>

<property name="userName" value="CICSUSER"/>

<property name="password" value="CICS"/>

</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">

<property name="managedConnectionFactory" ref="managedConnectionFactory"/>

</bean>

<bean id="component" class="MyDaoImpl">

<property name="connectionFactory" ref="connectionFactory"/>

</bean>

In managed mode (that is, in a Java EE environment), the configuration could look as follows:

<jee:jndi-lookup id="connectionFactory" jndi-name="eis/cicseci"/>

<bean id="component" class="MyDaoImpl">

<property name="connectionFactory" ref="connectionFactory"/>

</bean>

5.5. Transactions

JCA specifies several levels of transaction support for resource adapters. The kind of transactions that your resource adapter supports is specified in its ra.xml file. There are essentially three options: none (for example with CICS EPI connector), local transactions (for example with a CICS ECI connector), global transactions (for example with an IMS connector).

<connector>

<resourceadapter>

<!-- <transaction-support>NoTransaction</transaction-support> -->

<!-- <transaction-support>LocalTransaction</transaction-support> -->

<transaction-support>XATransaction</transaction-support>

<resourceadapter>

<connector>

For global transactions, you can use Spring’s generic transaction infrastructure to demarcate transactions, with JtaTransactionManager as backend (delegating to the Java EE server’s distributed transaction coordinator underneath).

For local transactions on a single CCI ConnectionFactory, Spring provides a specific transaction management strategy for CCI, analogous to the DataSourceTransactionManager for JDBC. The CCI API defines a local transaction object and corresponding local transaction demarcation methods. Spring’s CciLocalTransactionManager executes such local CCI transactions, fully compliant with Spring’s generic PlatformTransactionManager abstraction.

<jee:jndi-lookup id="eciConnectionFactory" jndi-name="eis/cicseci"/>

<bean id="eciTransactionManager"

class="org.springframework.jca.cci.connection.CciLocalTransactionManager">

<property name="connectionFactory" ref="eciConnectionFactory"/>

</bean>

Both transaction strategies can be used with any of Spring’s transaction demarcation facilities, be it declarative or programmatic. This is a consequence of Spring’s generic PlatformTransactionManager abstraction, which decouples transaction demarcation from the actual execution strategy. Simply switch between JtaTransactionManager and CciLocalTransactionManager as needed, keeping your transaction demarcation as-is.

For more information on Spring’s transaction facilities, see the chapter entitled Transaction Management.

6. Email

6.1. Introduction

Library dependencies

The following JAR needs to be on the classpath of your application in order to use the Spring Framework’s email library.

The JavaMail library

This library is freely available on the web — for example, in Maven Central as com.sun.mail:javax.mail.

The Spring Framework provides a helpful utility library for sending email that shields the user from the specifics of the underlying mailing system and is responsible for low level resource handling on behalf of the client.

The org.springframework.mail package is the root level package for the Spring Framework’s email support. The central interface for sending emails is the MailSender interface; a simple value object encapsulating the properties of a simple mail such as from and to (plus many others) is the SimpleMailMessage class. This package also contains a hierarchy of checked exceptions which provide a higher level of abstraction over the lower level mail system exceptions with the root exception being MailException. Please refer to the javadocs for more information on the rich mail exception hierarchy.

The org.springframework.mail.javamail.JavaMailSender interface adds specialized JavaMail features such as MIME message support to the MailSender interface (from which it inherits). JavaMailSender also provides a callback interface for preparing a 'MimeMessage', called org.springframework.mail.javamail.MimeMessagePreparator.

6.2. Usage

Let’s assume there is a business interface called OrderManager:

public interface OrderManager {

void placeOrder(Order order);

}

Let us also assume that there is a requirement stating that an email message with an order number needs to be generated and sent to a customer placing the relevant order.

6.2.1. Basic MailSender and SimpleMailMessage usage

import org.springframework.mail.MailException;

import org.springframework.mail.MailSender;

import org.springframework.mail.SimpleMailMessage;

public class SimpleOrderManager implements OrderManager {

private MailSender mailSender;

private SimpleMailMessage templateMessage;

public void setMailSender(MailSender mailSender) {

this.mailSender = mailSender;

}

public void setTemplateMessage(SimpleMailMessage templateMessage) {

this.templateMessage = templateMessage;

}

public void placeOrder(Order order) {

// Do the business calculations...

// Call the collaborators to persist the order...

// Create a thread safe "copy" of the template message and customize it

SimpleMailMessage msg = new SimpleMailMessage(this.templateMessage);

msg.setTo(order.getCustomer().getEmailAddress());

msg.setText(

"Dear " + order.getCustomer().getFirstName()

+ order.getCustomer().getLastName()

+ ", thank you for placing order. Your order number is "

+ order.getOrderNumber());

try{

this.mailSender.send(msg);

}

catch (MailException ex) {

// simply log it and go on...

System.err.println(ex.getMessage());

}

}

}

Find below the bean definitions for the above code:

<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">

<property name="host" value="mail.mycompany.com"/>

</bean>

<!-- this is a template message that we can pre-load with default state -->

<bean id="templateMessage" class="org.springframework.mail.SimpleMailMessage">

<property name="from" value="customerservice@mycompany.com"/>

<property name="subject" value="Your order"/>

</bean>

<bean id="orderManager" class="com.mycompany.businessapp.support.SimpleOrderManager">

<property name="mailSender" ref="mailSender"/>

<property name="templateMessage" ref="templateMessage"/>

</bean>

6.2.2. Using the JavaMailSender and the MimeMessagePreparator

Here is another implementation of OrderManager using the MimeMessagePreparator callback interface. Please note in this case that the mailSender property is of type JavaMailSender so that we are able to use the JavaMail MimeMessage class:

import javax.mail.Message;

import javax.mail.MessagingException;

import javax.mail.internet.InternetAddress;

import javax.mail.internet.MimeMessage;

import javax.mail.internet.MimeMessage;

import org.springframework.mail.MailException;

import org.springframework.mail.javamail.JavaMailSender;

import org.springframework.mail.javamail.MimeMessagePreparator;

public class SimpleOrderManager implements OrderManager {

private JavaMailSender mailSender;

public void setMailSender(JavaMailSender mailSender) {

this.mailSender = mailSender;

}

public void placeOrder(final Order order) {

// Do the business calculations...

// Call the collaborators to persist the order...

MimeMessagePreparator preparator = new MimeMessagePreparator() {

public void prepare(MimeMessage mimeMessage) throws Exception {

mimeMessage.setRecipient(Message.RecipientType.TO,

new InternetAddress(order.getCustomer().getEmailAddress()));

mimeMessage.setFrom(new InternetAddress("mail@mycompany.com"));

mimeMessage.setText("Dear " + order.getCustomer().getFirstName() + " " +

order.getCustomer().getLastName() + ", thanks for your order. " +

"Your order number is " + order.getOrderNumber() + ".");

}

};

try {

this.mailSender.send(preparator);

}

catch (MailException ex) {

// simply log it and go on...

System.err.println(ex.getMessage());

}

}

}

The mail code is a crosscutting concern and could well be a candidate for refactoring into a custom Spring AOP aspect, which then could be executed at appropriate joinpoints on the OrderManager target.

The Spring Framework’s mail support ships with the standard JavaMail implementation. Please refer to the relevant javadocs for more information.

6.3. Using the JavaMail MimeMessageHelper

A class that comes in pretty handy when dealing with JavaMail messages is the org.springframework.mail.javamail.MimeMessageHelper class, which shields you from having to use the verbose JavaMail API. Using the MimeMessageHelper it is pretty easy to create a MimeMessage:

// of course you would use DI in any real-world cases

JavaMailSenderImpl sender = new JavaMailSenderImpl();

sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

MimeMessageHelper helper = new MimeMessageHelper(message);

helper.setTo("test@host.com");

helper.setText("Thank you for ordering!");

sender.send(message);

6.3.1. Sending attachments and inline resources

Multipart email messages allow for both attachments and inline resources. Examples of inline resources would be images or a stylesheet you want to use in your message, but that you don’t want displayed as an attachment.

Attachments

The following example shows you how to use the MimeMessageHelper to send an email along with a single JPEG image attachment.

JavaMailSenderImpl sender = new JavaMailSenderImpl();

sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message

MimeMessageHelper helper = new MimeMessageHelper(message, true);

helper.setTo("test@host.com");

helper.setText("Check out this image!");

// let's attach the infamous windows Sample file (this time copied to c:/)

FileSystemResource file = new FileSystemResource(new File("c:/Sample.jpg"));

helper.addAttachment("CoolImage.jpg", file);

sender.send(message);

Inline resources

The following example shows you how to use the MimeMessageHelper to send an email along with an inline image.

JavaMailSenderImpl sender = new JavaMailSenderImpl();

sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message

MimeMessageHelper helper = new MimeMessageHelper(message, true);

helper.setTo("test@host.com");

// use the true flag to indicate the text included is HTML

helper.setText("<html><body><img src='cid:identifier1234'></body></html>", true);

// let's include the infamous windows Sample file (this time copied to c:/)

FileSystemResource res = new FileSystemResource(new File("c:/Sample.jpg"));

helper.addInline("identifier1234", res);

sender.send(message);

Inline resources are added to the MimeMessage using the specified Content-ID (identifier1234 in the above example). The order in which you are adding the text and the resource are very important. Be sure to first add the text and after that the resources. If you are doing it the other way around, it won’t work!

6.3.2. Creating email content using a templating library

The code in the previous examples explicitly created the content of the email message, using methods calls such as message.setText(..). This is fine for simple cases, and it is okay in the context of the aforementioned examples, where the intent was to show you the very basics of the API.

In your typical enterprise application though, you are not going to create the content of your emails using the above approach for a number of reasons.

Creating HTML-based email content in Java code is tedious and error prone

There is no clear separation between display logic and business logic

Changing the display structure of the email content requires writing Java code, recompiling, redeploying…​

Typically the approach taken to address these issues is to use a template library such as FreeMarker to define the display structure of email content. This leaves your code tasked only with creating the data that is to be rendered in the email template and sending the email. It is definitely a best practice for when the content of your emails becomes even moderately complex, and with the Spring Framework’s support classes for FreeMarker becomes quite easy to do.

7. Task Execution and Scheduling

7.1. Introduction

The Spring Framework provides abstractions for asynchronous execution and scheduling of tasks with the TaskExecutor and TaskScheduler interfaces, respectively. Spring also features implementations of those interfaces that support thread pools or delegation to CommonJ within an application server environment. Ultimately the use of these implementations behind the common interfaces abstracts away the differences between Java SE 5, Java SE 6 and Java EE environments.

Spring also features integration classes for supporting scheduling with the Timer, part of the JDK since 1.3, and the Quartz Scheduler ( http://quartz-scheduler.org). Both of those schedulers are set up using a FactoryBean with optional references to Timer or Trigger instances, respectively. Furthermore, a convenience class for both the Quartz Scheduler and the Timer is available that allows you to invoke a method of an existing target object (analogous to the normal MethodInvokingFactoryBean operation).

7.2. The Spring TaskExecutor abstraction

Executors are the JDK name for the concept of thread pools. The "executor" naming is due to the fact that there is no guarantee that the underlying implementation is actually a pool; an executor may be single-threaded or even synchronous. Spring’s abstraction hides implementation details between Java SE and Java EE environments.

Spring’s TaskExecutor interface is identical to the java.util.concurrent.Executor interface. In fact, originally, its primary reason for existence was to abstract away the need for Java 5 when using thread pools. The interface has a single method execute(Runnable task) that accepts a task for execution based on the semantics and configuration of the thread pool.

The TaskExecutor was originally created to give other Spring components an abstraction for thread pooling where needed. Components such as the ApplicationEventMulticaster, JMS’s AbstractMessageListenerContainer, and Quartz integration all use the TaskExecutor abstraction to pool threads. However, if your beans need thread pooling behavior, it is possible to use this abstraction for your own needs.

7.2.1. TaskExecutor types

There are a number of pre-built implementations of TaskExecutor included with the Spring distribution. In all likelihood, you should never need to implement your own. The common out-of-the-box variants are:

SyncTaskExecutor This implementation does not execute invocations asynchronously. Instead, each invocation takes place in the calling thread. It is primarily used in situations where multi-threading is not necessary such as in simple test cases.

SimpleAsyncTaskExecutor This implementation does not reuse any threads, rather it starts up a new thread for each invocation. However, it does support a concurrency limit which will block any invocations that are over the limit until a slot has been freed up. If you are looking for true pooling, see ThreadPoolTaskExecutor below.

ConcurrentTaskExecutor This implementation is an adapter for a java.util.concurrent.Executor instance. There is an alternative, ThreadPoolTaskExecutor, that exposes the Executor configuration parameters as bean properties. There is rarely a need to use ConcurrentTaskExecutor directly, but if the ThreadPoolTaskExecutor is not flexible enough for your needs, then ConcurrentTaskExecutor is an alternative.

ThreadPoolTaskExecutor This implementation is the most commonly used one. It exposes bean properties for configuring a java.util.concurrent.ThreadPoolExecutor and wraps it in a TaskExecutor. If you need to adapt to a different kind of java.util.concurrent.Executor, it is recommended that you use a ConcurrentTaskExecutor instead.

WorkManagerTaskExecutor This implementation uses a CommonJ WorkManager as its backing service provider and is the central convenience class for setting up CommonJ-based thread pool integration on WebLogic/WebSphere within a Spring application context.

DefaultManagedTaskExecutor This implementation uses a JNDI-obtained ManagedExecutorService in a JSR-236 compatible runtime environment such as a Java EE 7+ application server, replacing a CommonJ WorkManager for that purpose.

7.2.2. Using a TaskExecutor

Spring’s TaskExecutor implementations are used as simple JavaBeans. In the example below, we define a bean that uses the ThreadPoolTaskExecutor to asynchronously print out a set of messages.

import org.springframework.core.task.TaskExecutor;

public class TaskExecutorExample {

private class MessagePrinterTask implements Runnable {

private String message;

public MessagePrinterTask(String message) {

this.message = message;

}

public void run() {

System.out.println(message);

}

}

private TaskExecutor taskExecutor;

public TaskExecutorExample(TaskExecutor taskExecutor) {

this.taskExecutor = taskExecutor;

}

public void printMessages() {

for(int i = 0; i < 25; i++) {

taskExecutor.execute(new MessagePrinterTask("Message" + i));

}

}

}

As you can see, rather than retrieving a thread from the pool and executing yourself, you add your Runnable to the queue and the TaskExecutor uses its internal rules to decide when the task gets executed.

To configure the rules that the TaskExecutor will use, simple bean properties have been exposed.

<bean id="taskExecutor" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">

<property name="corePoolSize" value="5"/>

<property name="maxPoolSize" value="10"/>

<property name="queueCapacity" value="25"/>

</bean>

<bean id="taskExecutorExample" class="TaskExecutorExample">

<constructor-arg ref="taskExecutor"/>

</bean>

7.3. The Spring TaskScheduler abstraction

In addition to the TaskExecutor abstraction, Spring 3.0 introduces a TaskScheduler with a variety of methods for scheduling tasks to run at some point in the future.

public interface TaskScheduler {

ScheduledFuture schedule(Runnable task, Trigger trigger);

ScheduledFuture schedule(Runnable task, Instant startTime);

ScheduledFuture schedule(Runnable task, Date startTime);

ScheduledFuture scheduleAtFixedRate(Runnable task, Instant startTime, Duration period);

ScheduledFuture scheduleAtFixedRate(Runnable task, Date startTime, long period);

ScheduledFuture scheduleAtFixedRate(Runnable task, Duration period);

ScheduledFuture scheduleAtFixedRate(Runnable task, long period);

ScheduledFuture scheduleWithFixedDelay(Runnable task, Instant startTime, Duration delay);

ScheduledFuture scheduleWithFixedDelay(Runnable task, Date startTime, long delay);

ScheduledFuture scheduleWithFixedDelay(Runnable task, Duration delay);

ScheduledFuture scheduleWithFixedDelay(Runnable task, long delay);

}

The simplest method is the one named 'schedule' that takes a Runnable and Date only. That will cause the task to run once after the specified time. All of the other methods are capable of scheduling tasks to run repeatedly. The fixed-rate and fixed-delay methods are for simple, periodic execution, but the method that accepts a Trigger is much more flexible.

7.3.1. Trigger interface

The Trigger interface is essentially inspired by JSR-236 which, as of Spring 3.0, was not yet officially implemented. The basic idea of the Trigger is that execution times may be determined based on past execution outcomes or even arbitrary conditions. If these determinations do take into account the outcome of the preceding execution, that information is available within a TriggerContext. The Trigger interface itself is quite simple:

public interface Trigger {

Date nextExecutionTime(TriggerContext triggerContext);

}

As you can see, the TriggerContext is the most important part. It encapsulates all of the relevant data, and is open for extension in the future if necessary. The TriggerContext is an interface (a SimpleTriggerContext implementation is used by default). Here you can see what methods are available for Trigger implementations.

public interface TriggerContext {

Date lastScheduledExecutionTime();

Date lastActualExecutionTime();

Date lastCompletionTime();

}

7.3.2. Trigger implementations

Spring provides two implementations of the Trigger interface. The most interesting one is the CronTrigger. It enables the scheduling of tasks based on cron expressions. For example, the following task is being scheduled to run 15 minutes past each hour but only during the 9-to-5 "business hours" on weekdays.

scheduler.schedule(task, new CronTrigger("0 15 9-17 \* \* MON-FRI"));

The other out-of-the-box implementation is a PeriodicTrigger that accepts a fixed period, an optional initial delay value, and a boolean to indicate whether the period should be interpreted as a fixed-rate or a fixed-delay. Since the TaskScheduler interface already defines methods for scheduling tasks at a fixed-rate or with a fixed-delay, those methods should be used directly whenever possible. The value of the PeriodicTrigger implementation is that it can be used within components that rely on the Trigger abstraction. For example, it may be convenient to allow periodic triggers, cron-based triggers, and even custom trigger implementations to be used interchangeably. Such a component could take advantage of dependency injection so that such Triggers could be configured externally and therefore easily modified or extended.

7.3.3. TaskScheduler implementations

As with Spring’s TaskExecutor abstraction, the primary benefit of the TaskScheduler arrangement is that an application’s scheduling needs are decoupled from the deployment environment. This abstraction level is particularly relevant when deploying to an application server environment where threads should not be created directly by the application itself. For such scenarios, Spring provides a TimerManagerTaskScheduler delegating to a CommonJ TimerManager on WebLogic/WebSphere as well as a more recent DefaultManagedTaskScheduler delegating to a JSR-236 ManagedScheduledExecutorService in a Java EE 7+ environment, both typically configured with a JNDI lookup.

Whenever external thread management is not a requirement, a simpler alternative is a local ScheduledExecutorService setup within the application which can be adapted through Spring’s ConcurrentTaskScheduler. As a convenience, Spring also provides a ThreadPoolTaskScheduler which internally delegates to a ScheduledExecutorService, providing common bean-style configuration along the lines of ThreadPoolTaskExecutor. These variants work perfectly fine for locally embedded thread pool setups in lenient application server environments as well, in particular on Tomcat and Jetty.

7.4. Annotation Support for Scheduling and Asynchronous Execution

Spring provides annotation support for both task scheduling and asynchronous method execution.

7.4.1. Enable scheduling annotations

To enable support for @Scheduled and @Async annotations add @EnableScheduling and @EnableAsync to one of your @Configuration classes:

@Configuration

@EnableAsync

@EnableScheduling

public class AppConfig {

}

You are free to pick and choose the relevant annotations for your application. For example, if you only need support for @Scheduled, simply omit @EnableAsync. For more fine-grained control you can additionally implement the SchedulingConfigurer and/or AsyncConfigurer interfaces. See the javadocs for full details.

If you prefer XML configuration use the <task:annotation-driven> element.

<task:annotation-driven executor="myExecutor" scheduler="myScheduler"/>

<task:executor id="myExecutor" pool-size="5"/>

<task:scheduler id="myScheduler" pool-size="10"/>

Notice with the above XML that an executor reference is provided for handling those tasks that correspond to methods with the @Async annotation, and the scheduler reference is provided for managing those methods annotated with @Scheduled.

The default advice mode for processing @Async annotations is "proxy" which allows for interception of calls through the proxy only; local calls within the same class cannot get intercepted that way. For a more advanced mode of interception, consider switching to "aspectj" mode in combination with compile-time or load-time weaving.

7.4.2. The @Scheduled annotation

The @Scheduled annotation can be added to a method along with trigger metadata. For example, the following method would be invoked every 5 seconds with a fixed delay, meaning that the period will be measured from the completion time of each preceding invocation.

@Scheduled(fixedDelay=5000)

public void doSomething() {

// something that should execute periodically

}

If a fixed rate execution is desired, simply change the property name specified within the annotation. The following would be executed every 5 seconds measured between the successive start times of each invocation.

@Scheduled(fixedRate=5000)

public void doSomething() {

// something that should execute periodically

}

For fixed-delay and fixed-rate tasks, an initial delay may be specified indicating the number of milliseconds to wait before the first execution of the method.

@Scheduled(initialDelay=1000, fixedRate=5000)

public void doSomething() {

// something that should execute periodically

}

If simple periodic scheduling is not expressive enough, then a cron expression may be provided. For example, the following will only execute on weekdays.

@Scheduled(cron="\*/5 \* \* \* \* MON-FRI")

public void doSomething() {

// something that should execute on weekdays only

}

You can additionally use the zone attribute to specify the time zone in which the cron expression will be resolved.

Notice that the methods to be scheduled must have void returns and must not expect any arguments. If the method needs to interact with other objects from the Application Context, then those would typically have been provided through dependency injection.

As of Spring Framework 4.3, @Scheduled methods are supported on beans of any scope.

Make sure that you are not initializing multiple instances of the same @Scheduled annotation class at runtime, unless you do want to schedule callbacks to each such instance. Related to this, make sure that you do not use @Configurable on bean classes which are annotated with @Scheduled and registered as regular Spring beans with the container: You would get double initialization otherwise, once through the container and once through the @Configurable aspect, with the consequence of each @Scheduled method being invoked twice.

7.4.3. The @Async annotation

The @Async annotation can be provided on a method so that invocation of that method will occur asynchronously. In other words, the caller will return immediately upon invocation and the actual execution of the method will occur in a task that has been submitted to a Spring TaskExecutor. In the simplest case, the annotation may be applied to a void-returning method.

@Async

void doSomething() {

// this will be executed asynchronously

}

Unlike the methods annotated with the @Scheduled annotation, these methods can expect arguments, because they will be invoked in the "normal" way by callers at runtime rather than from a scheduled task being managed by the container. For example, the following is a legitimate application of the @Async annotation.

@Async

void doSomething(String s) {

// this will be executed asynchronously

}

Even methods that return a value can be invoked asynchronously. However, such methods are required to have a Future typed return value. This still provides the benefit of asynchronous execution so that the caller can perform other tasks prior to calling get() on that Future.

@Async

Future<String> returnSomething(int i) {

// this will be executed asynchronously

}

@Async methods may not only declare a regular java.util.concurrent.Future return type but also Spring’s org.springframework.util.concurrent.ListenableFuture or, as of Spring 4.2, JDK 8’s java.util.concurrent.CompletableFuture: for richer interaction with the asynchronous task and for immediate composition with further processing steps.

@Async can not be used in conjunction with lifecycle callbacks such as @PostConstruct. To asynchronously initialize Spring beans you currently have to use a separate initializing Spring bean that invokes the @Async annotated method on the target then.

public class SampleBeanImpl implements SampleBean {

@Async

void doSomething() {

// ...

}

}

public class SampleBeanInitializer {

private final SampleBean bean;

public SampleBeanInitializer(SampleBean bean) {

this.bean = bean;

}

@PostConstruct

public void initialize() {

bean.doSomething();

}

}

There is no direct XML equivalent for @Async since such methods should be designed for asynchronous execution in the first place, not externally re-declared to be async. However, you may manually set up Spring’s AsyncExecutionInterceptor with Spring AOP, in combination with a custom pointcut.

7.4.4. Executor qualification with @Async

By default when specifying @Async on a method, the executor that will be used is the one supplied to the 'annotation-driven' element as described above. However, the value attribute of the @Async annotation can be used when needing to indicate that an executor other than the default should be used when executing a given method.

@Async("otherExecutor")

void doSomething(String s) {

// this will be executed asynchronously by "otherExecutor"

}

In this case, "otherExecutor" may be the name of any Executor bean in the Spring container, or may be the name of a qualifier associated with any Executor, e.g. as specified with the <qualifier> element or Spring’s @Qualifier annotation.

7.4.5. Exception management with @Async

When an @Async method has a Future typed return value, it is easy to manage an exception that was thrown during the method execution as this exception will be thrown when calling get on the Future result. With a void return type however, the exception is uncaught and cannot be transmitted. For those cases, an AsyncUncaughtExceptionHandler can be provided to handle such exceptions.

public class MyAsyncUncaughtExceptionHandler implements AsyncUncaughtExceptionHandler {

@Override

public void handleUncaughtException(Throwable ex, Method method, Object... params) {

// handle exception

}

}

By default, the exception is simply logged. A custom AsyncUncaughtExceptionHandler can be defined via AsyncConfigurer or the task:annotation-driven XML element.

7.5. The task namespace

Beginning with Spring 3.0, there is an XML namespace for configuring TaskExecutor and TaskScheduler instances. It also provides a convenient way to configure tasks to be scheduled with a trigger.

7.5.1. The 'scheduler' element

The following element will create a ThreadPoolTaskScheduler instance with the specified thread pool size.

<task:scheduler id="scheduler" pool-size="10"/>

The value provided for the 'id' attribute will be used as the prefix for thread names within the pool. The 'scheduler' element is relatively straightforward. If you do not provide a 'pool-size' attribute, the default thread pool will only have a single thread. There are no other configuration options for the scheduler.

7.5.2. The 'executor' element

The following will create a ThreadPoolTaskExecutor instance:

<task:executor id="executor" pool-size="10"/>

As with the scheduler above, the value provided for the 'id' attribute will be used as the prefix for thread names within the pool. As far as the pool size is concerned, the 'executor' element supports more configuration options than the 'scheduler' element. For one thing, the thread pool for a ThreadPoolTaskExecutor is itself more configurable. Rather than just a single size, an executor’s thread pool may have different values for the core and the max size. If a single value is provided then the executor will have a fixed-size thread pool (the core and max sizes are the same). However, the 'executor' element’s 'pool-size' attribute also accepts a range in the form of "min-max".

<task:executor

id="executorWithPoolSizeRange"

pool-size="5-25"

queue-capacity="100"/>

As you can see from that configuration, a 'queue-capacity' value has also been provided. The configuration of the thread pool should also be considered in light of the executor’s queue capacity. For the full description of the relationship between pool size and queue capacity, consult the documentation for ThreadPoolExecutor. The main idea is that when a task is submitted, the executor will first try to use a free thread if the number of active threads is currently less than the core size. If the core size has been reached, then the task will be added to the queue as long as its capacity has not yet been reached. Only then, if the queue’s capacity has been reached, will the executor create a new thread beyond the core size. If the max size has also been reached, then the executor will reject the task.

By default, the queue is unbounded, but this is rarely the desired configuration, because it can lead to OutOfMemoryErrors if enough tasks are added to that queue while all pool threads are busy. Furthermore, if the queue is unbounded, then the max size has no effect at all. Since the executor will always try the queue before creating a new thread beyond the core size, a queue must have a finite capacity for the thread pool to grow beyond the core size (this is why a fixed size pool is the only sensible case when using an unbounded queue).

In a moment, we will review the effects of the keep-alive setting which adds yet another factor to consider when providing a pool size configuration. First, let’s consider the case, as mentioned above, when a task is rejected. By default, when a task is rejected, a thread pool executor will throw a TaskRejectedException. However, the rejection policy is actually configurable. The exception is thrown when using the default rejection policy which is the AbortPolicy implementation. For applications where some tasks can be skipped under heavy load, either the DiscardPolicy or DiscardOldestPolicy may be configured instead. Another option that works well for applications that need to throttle the submitted tasks under heavy load is the CallerRunsPolicy. Instead of throwing an exception or discarding tasks, that policy will simply force the thread that is calling the submit method to run the task itself. The idea is that such a caller will be busy while running that task and not able to submit other tasks immediately. Therefore it provides a simple way to throttle the incoming load while maintaining the limits of the thread pool and queue. Typically this allows the executor to "catch up" on the tasks it is handling and thereby frees up some capacity on the queue, in the pool, or both. Any of these options can be chosen from an enumeration of values available for the 'rejection-policy' attribute on the 'executor' element.

<task:executor

id="executorWithCallerRunsPolicy"

pool-size="5-25"

queue-capacity="100"

rejection-policy="CALLER\_RUNS"/>

Finally, the keep-alive setting determines the time limit (in seconds) for which threads may remain idle before being terminated. If there are more than the core number of threads currently in the pool, after waiting this amount of time without processing a task, excess threads will get terminated. A time value of zero will cause excess threads to terminate immediately after executing a task without remaining follow-up work in the task queue.

<task:executor

id="executorWithKeepAlive"

pool-size="5-25"

keep-alive="120"/>

7.5.3. The 'scheduled-tasks' element

The most powerful feature of Spring’s task namespace is the support for configuring tasks to be scheduled within a Spring Application Context. This follows an approach similar to other "method-invokers" in Spring, such as that provided by the JMS namespace for configuring Message-driven POJOs. Basically a "ref" attribute can point to any Spring-managed object, and the "method" attribute provides the name of a method to be invoked on that object. Here is a simple example.

<task:scheduled-tasks scheduler="myScheduler">

<task:scheduled ref="beanA" method="methodA" fixed-delay="5000"/>

</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="10"/>

As you can see, the scheduler is referenced by the outer element, and each individual task includes the configuration of its trigger metadata. In the preceding example, that metadata defines a periodic trigger with a fixed delay indicating the number of milliseconds to wait after each task execution has completed. Another option is 'fixed-rate', indicating how often the method should be executed regardless of how long any previous execution takes. Additionally, for both fixed-delay and fixed-rate tasks an 'initial-delay' parameter may be specified indicating the number of milliseconds to wait before the first execution of the method. For more control, a "cron" attribute may be provided instead. Here is an example demonstrating these other options.

<task:scheduled-tasks scheduler="myScheduler">

<task:scheduled ref="beanA" method="methodA" fixed-delay="5000" initial-delay="1000"/>

<task:scheduled ref="beanB" method="methodB" fixed-rate="5000"/>

<task:scheduled ref="beanC" method="methodC" cron="\*/5 \* \* \* \* MON-FRI"/>

</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="10"/>

7.6. Using the Quartz Scheduler

Quartz uses Trigger, Job and JobDetail objects to realize scheduling of all kinds of jobs. For the basic concepts behind Quartz, have a look at http://quartz-scheduler.org. For convenience purposes, Spring offers a couple of classes that simplify the usage of Quartz within Spring-based applications.

7.6.1. Using the JobDetailFactoryBean

Quartz JobDetail objects contain all information needed to run a job. Spring provides a JobDetailFactoryBean which provides bean-style properties for XML configuration purposes. Let’s have a look at an example:

<bean name="exampleJob" class="org.springframework.scheduling.quartz.JobDetailFactoryBean">

<property name="jobClass" value="example.ExampleJob"/>

<property name="jobDataAsMap">

<map>

<entry key="timeout" value="5"/>

</map>

</property>

</bean>

The job detail configuration has all information it needs to run the job (ExampleJob). The timeout is specified in the job data map. The job data map is available through the JobExecutionContext (passed to you at execution time), but the JobDetail also gets its properties from the job data mapped to properties of the job instance. So in this case, if the ExampleJob contains a bean property named timeout, the JobDetail will have it applied automatically:

package example;

public class ExampleJob extends QuartzJobBean {

private int timeout;

/\*\*

\* Setter called after the ExampleJob is instantiated

\* with the value from the JobDetailFactoryBean (5)

\*/

public void setTimeout(int timeout) {

this.timeout = timeout;

}

protected void executeInternal(JobExecutionContext ctx) throws JobExecutionException {

// do the actual work

}

}

All additional properties from the job data map are of course available to you as well.

Using the name and group properties, you can modify the name and the group of the job, respectively. By default, the name of the job matches the bean name of the JobDetailFactoryBean (in the example above, this is exampleJob).

7.6.2. Using the MethodInvokingJobDetailFactoryBean

Often you just need to invoke a method on a specific object. Using the MethodInvokingJobDetailFactoryBean you can do exactly this:

<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">

<property name="targetObject" ref="exampleBusinessObject"/>

<property name="targetMethod" value="doIt"/>

</bean>

The above example will result in the doIt method being called on the exampleBusinessObject method (see below):

public class ExampleBusinessObject {

// properties and collaborators

public void doIt() {

// do the actual work

}

}

<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>

Using the MethodInvokingJobDetailFactoryBean, you don’t need to create one-line jobs that just invoke a method, and you only need to create the actual business object and wire up the detail object.

By default, Quartz Jobs are stateless, resulting in the possibility of jobs interfering with each other. If you specify two triggers for the same JobDetail, it might be possible that before the first job has finished, the second one will start. If JobDetail classes implement the Stateful interface, this won’t happen. The second job will not start before the first one has finished. To make jobs resulting from the MethodInvokingJobDetailFactoryBean non-concurrent, set the concurrent flag to false.

<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">

<property name="targetObject" ref="exampleBusinessObject"/>

<property name="targetMethod" value="doIt"/>

<property name="concurrent" value="false"/>

</bean>

By default, jobs will run in a concurrent fashion.

7.6.3. Wiring up jobs using triggers and the SchedulerFactoryBean

We’ve created job details and jobs. We’ve also reviewed the convenience bean that allows you to invoke a method on a specific object. Of course, we still need to schedule the jobs themselves. This is done using triggers and a SchedulerFactoryBean. Several triggers are available within Quartz and Spring offers two Quartz FactoryBean implementations with convenient defaults: CronTriggerFactoryBean and SimpleTriggerFactoryBean.

Triggers need to be scheduled. Spring offers a SchedulerFactoryBean that exposes triggers to be set as properties. SchedulerFactoryBean schedules the actual jobs with those triggers.

Find below a couple of examples:

<bean id="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerFactoryBean">

<!-- see the example of method invoking job above -->

<property name="jobDetail" ref="jobDetail"/>

<!-- 10 seconds -->

<property name="startDelay" value="10000"/>

<!-- repeat every 50 seconds -->

<property name="repeatInterval" value="50000"/>

</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerFactoryBean">

<property name="jobDetail" ref="exampleJob"/>

<!-- run every morning at 6 AM -->

<property name="cronExpression" value="0 0 6 \* \* ?"/>

</bean>

Now we’ve set up two triggers, one running every 50 seconds with a starting delay of 10 seconds and one every morning at 6 AM. To finalize everything, we need to set up the SchedulerFactoryBean:

<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">

<property name="triggers">

<list>

<ref bean="cronTrigger"/>

<ref bean="simpleTrigger"/>

</list>

</property>

</bean>

More properties are available for the SchedulerFactoryBean for you to set, such as the calendars used by the job details, properties to customize Quartz with, etc. Have a look at the SchedulerFactoryBean javadocs for more information.

8. Cache Abstraction

8.1. Introduction

Since version 3.1, Spring Framework provides support for transparently adding caching into an existing Spring application. Similar to the transaction support, the caching abstraction allows consistent use of various caching solutions with minimal impact on the code.

As from Spring 4.1, the cache abstraction has been significantly improved with the support of JSR-107 annotations and more customization options.

8.2. Understanding the cache abstraction

Cache vs Buffer

The terms "buffer" and "cache" tend to be used interchangeably; note however they represent different things. A buffer is used traditionally as an intermediate temporary store for data between a fast and a slow entity. As one party would have to wait for the other affecting performance, the buffer alleviates this by allowing entire blocks of data to move at once rather then in small chunks. The data is written and read only once from the buffer. Furthermore, the buffers are visible to at least one party which is aware of it.

A cache on the other hand by definition is hidden and neither party is aware that caching occurs.It as well improves performance but does that by allowing the same data to be read multiple times in a fast fashion.

A further explanation of the differences between two can be found here.

At its core, the abstraction applies caching to Java methods, reducing thus the number of executions based on the information available in the cache. That is, each time a targeted method is invoked, the abstraction will apply a caching behavior checking whether the method has been already executed for the given arguments. If it has, then the cached result is returned without having to execute the actual method; if it has not, then method is executed, the result cached and returned to the user so that, the next time the method is invoked, the cached result is returned. This way, expensive methods (whether CPU or IO bound) can be executed only once for a given set of parameters and the result reused without having to actually execute the method again. The caching logic is applied transparently without any interference to the invoker.

Obviously this approach works only for methods that are guaranteed to return the same output (result) for a given input (or arguments) no matter how many times it is being executed.

Other cache-related operations are provided by the abstraction such as the ability to update the content of the cache or remove one of all entries. These are useful if the cache deals with data that can change during the course of the application.

Just like other services in the Spring Framework, the caching service is an abstraction (not a cache implementation) and requires the use of an actual storage to store the cache data - that is, the abstraction frees the developer from having to write the caching logic but does not provide the actual stores. This abstraction is materialized by the org.springframework.cache.Cache and org.springframework.cache.CacheManager interfaces.

There are a few implementations of that abstraction available out of the box: JDK java.util.concurrent.ConcurrentMap based caches, Ehcache 2.x, Gemfire cache, Caffeine and JSR-107 compliant caches (e.g. Ehcache 3.x). See Plugging-in different back-end caches for more information on plugging in other cache stores/providers.

The caching abstraction has no special handling of multi-threaded and multi-process environments as such features are handled by the cache implementation. .

If you have a multi-process environment (i.e. an application deployed on several nodes), you will need to configure your cache provider accordingly. Depending on your use cases, a copy of the same data on several nodes may be enough but if you change the data during the course of the application, you may need to enable other propagation mechanisms.

Caching a particular item is a direct equivalent of the typical get-if-not-found-then- proceed-and-put-eventually code blocks found with programmatic cache interaction: no locks are applied and several threads may try to load the same item concurrently. The same applies to eviction: if several threads are trying to update or evict data concurrently, you may use stale data. Certain cache providers offer advanced features in that area, refer to the documentation of the cache provider that you are using for more details.

To use the cache abstraction, the developer needs to take care of two aspects:

caching declaration - identify the methods that need to be cached and their policy

cache configuration - the backing cache where the data is stored and read from

8.3. Declarative annotation-based caching

For caching declaration, the abstraction provides a set of Java annotations:

@Cacheable triggers cache population

@CacheEvict triggers cache eviction

@CachePut updates the cache without interfering with the method execution

@Caching regroups multiple cache operations to be applied on a method

@CacheConfig shares some common cache-related settings at class-level

Let us take a closer look at each annotation:

8.3.1. @Cacheable annotation

As the name implies, @Cacheable is used to demarcate methods that are cacheable - that is, methods for whom the result is stored into the cache so on subsequent invocations (with the same arguments), the value in the cache is returned without having to actually execute the method. In its simplest form, the annotation declaration requires the name of the cache associated with the annotated method:

@Cacheable("books")

public Book findBook(ISBN isbn) {...}

In the snippet above, the method findBook is associated with the cache named books. Each time the method is called, the cache is checked to see whether the invocation has been already executed and does not have to be repeated. While in most cases, only one cache is declared, the annotation allows multiple names to be specified so that more than one cache are being used. In this case, each of the caches will be checked before executing the method - if at least one cache is hit, then the associated value will be returned:

All the other caches that do not contain the value will be updated as well even though the cached method was not actually executed.

@Cacheable({"books", "isbns"})

public Book findBook(ISBN isbn) {...}

Default Key Generation

Since caches are essentially key-value stores, each invocation of a cached method needs to be translated into a suitable key for cache access. Out of the box, the caching abstraction uses a simple KeyGenerator based on the following algorithm:

If no params are given, return SimpleKey.EMPTY.

If only one param is given, return that instance.

If more the one param is given, return a SimpleKey containing all parameters.

This approach works well for most use-cases; As long as parameters have natural keys and implement valid hashCode() and equals() methods. If that is not the case then the strategy needs to be changed.

To provide a different default key generator, one needs to implement the org.springframework.cache.interceptor.KeyGenerator interface.

The default key generation strategy changed with the release of Spring 4.0. Earlier versions of Spring used a key generation strategy that, for multiple key parameters, only considered the hashCode() of parameters and not equals(); this could cause unexpected key collisions (see SPR-10237 for background). The new 'SimpleKeyGenerator' uses a compound key for such scenarios.

If you want to keep using the previous key strategy, you can configure the deprecated org.springframework.cache.interceptor.DefaultKeyGenerator class or create a custom hash-based 'KeyGenerator' implementation.

Custom Key Generation Declaration

Since caching is generic, it is quite likely the target methods have various signatures that cannot be simply mapped on top of the cache structure. This tends to become obvious when the target method has multiple arguments out of which only some are suitable for caching (while the rest are used only by the method logic). For example:

@Cacheable("books")

public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

At first glance, while the two boolean arguments influence the way the book is found, they are no use for the cache. Further more what if only one of the two is important while the other is not?

For such cases, the @Cacheable annotation allows the user to specify how the key is generated through its key attribute. The developer can use SpEL to pick the arguments of interest (or their nested properties), perform operations or even invoke arbitrary methods without having to write any code or implement any interface. This is the recommended approach over the default generator since methods tend to be quite different in signatures as the code base grows; while the default strategy might work for some methods, it rarely does for all methods.

Below are some examples of various SpEL declarations - if you are not familiar with it, do yourself a favor and read Spring Expression Language:

@Cacheable(cacheNames="books", key="#isbn")

public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(cacheNames="books", key="#isbn.rawNumber")

public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(cacheNames="books", key="T(someType).hash(#isbn)")

public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

The snippets above show how easy it is to select a certain argument, one of its properties or even an arbitrary (static) method.

If the algorithm responsible to generate the key is too specific or if it needs to be shared, you may define a custom keyGenerator on the operation. To do this, specify the name of the KeyGenerator bean implementation to use:

@Cacheable(cacheNames="books", keyGenerator="myKeyGenerator")

public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

The key and keyGenerator parameters are mutually exclusive and an operation specifying both will result in an exception.

Default Cache Resolution

Out of the box, the caching abstraction uses a simple CacheResolver that retrieves the cache(s) defined at the operation level using the configured CacheManager.

To provide a different default cache resolver, one needs to implement the org.springframework.cache.interceptor.CacheResolver interface.

Custom cache resolution

The default cache resolution fits well for applications working with a single CacheManager and with no complex cache resolution requirements.

For applications working with several cache managers, it is possible to set the cacheManager to use per operation:

@Cacheable(cacheNames="books", cacheManager="anotherCacheManager")

public Book findBook(ISBN isbn) {...}

It is also possible to replace the CacheResolver entirely in a similar fashion as for key generation. The resolution is requested for every cache operation, giving a chance to the implementation to actually resolve the cache(s) to use based on runtime arguments:

@Cacheable(cacheResolver="runtimeCacheResolver")

public Book findBook(ISBN isbn) {...}

Since Spring 4.1, the value attribute of the cache annotations are no longer mandatory since this particular information can be provided by the CacheResolver regardless of the content of the annotation.

Similarly to key and keyGenerator, the cacheManager and cacheResolver parameters are mutually exclusive and an operation specifying both will result in an exception as a custom CacheManager will be ignored by the CacheResolver implementation. This is probably not what you expect.

Synchronized caching

In a multi-threaded environment, certain operations might be concurrently invoked for the same argument (typically on startup). By default, the cache abstraction does not lock anything and the same value may be computed several times, defeating the purpose of caching.

For those particular cases, the sync attribute can be used to instruct the underlying cache provider to lock the cache entry while the value is being computed. As a result, only one thread will be busy computing the value while the others are blocked until the entry is updated in the cache.

@Cacheable(cacheNames="foos", sync=true)

public Foo executeExpensiveOperation(String id) {...}

This is an optional feature and your favorite cache library may not support it. All CacheManager implementations provided by the core framework support it. Check the documentation of your cache provider for more details.

Conditional caching

Sometimes, a method might not be suitable for caching all the time (for example, it might depend on the given arguments). The cache annotations support such functionality through the condition parameter which takes a SpEL expression that is evaluated to either true or false. If true, the method is cached - if not, it behaves as if the method is not cached, that is executed every time no matter what values are in the cache or what arguments are used. A quick example - the following method will be cached only if the argument name has a length shorter than 32:

@Cacheable(cacheNames="book", condition="#name.length() < 32")

public Book findBook(String name)

In addition the condition parameter, the unless parameter can be used to veto the adding of a value to the cache. Unlike condition, unless expressions are evaluated after the method has been called. Expanding on the previous example - perhaps we only want to cache paperback books:

@Cacheable(cacheNames="book", condition="#name.length() < 32", unless="#result.hardback")

public Book findBook(String name)

The cache abstraction supports java.util.Optional, using its content as cached value only if it present. #result always refers to the business entity and never on a supported wrapper so the previous example can be rewritten as follows:

@Cacheable(cacheNames="book", condition="#name.length() < 32", unless="#result?.hardback")

public Optional<Book> findBook(String name)

Note that result still refers to Book and not Optional. As it might be null, we should use the safe navigation operator.

Available caching SpEL evaluation context

Each SpEL expression evaluates again a dedicated context. In addition to the build in parameters, the framework provides dedicated caching related metadata such as the argument names. The next table lists the items made available to the context so one can use them for key and conditional computations:

Table 11. Cache SpEL available metadata

Name Location Description Example

methodName

root object

The name of the method being invoked

#root.methodName

method

root object

The method being invoked

#root.method.name

target

root object

The target object being invoked

#root.target

targetClass

root object

The class of the target being invoked

#root.targetClass

args

root object

The arguments (as array) used for invoking the target

#root.args[0]

caches

root object

Collection of caches against which the current method is executed

#root.caches[0].name

argument name

evaluation context

Name of any of the method arguments. If for some reason the names are not available (e.g. no debug information), the argument names are also available under the #a<#arg> where #arg stands for the argument index (starting from 0).

#iban or #a0 (one can also use #p0 or #p<#arg> notation as an alias).

result

evaluation context

The result of the method call (the value to be cached). Only available in unless expressions, cache put expressions (to compute the key), or cache evict expressions (when beforeInvocation is false). For supported wrappers such as Optional, #result refers to the actual object, not the wrapper.

#result

8.3.2. @CachePut annotation

For cases where the cache needs to be updated without interfering with the method execution, one can use the @CachePut annotation. That is, the method will always be executed and its result placed into the cache (according to the @CachePut options). It supports the same options as @Cacheable and should be used for cache population rather than method flow optimization:

@CachePut(cacheNames="book", key="#isbn")

public Book updateBook(ISBN isbn, BookDescriptor descriptor)

Note that using @CachePut and @Cacheable annotations on the same method is generally strongly discouraged because they have different behaviors. While the latter causes the method execution to be skipped by using the cache, the former forces the execution in order to execute a cache update. This leads to unexpected behavior and with the exception of specific corner-cases (such as annotations having conditions that exclude them from each other), such declaration should be avoided. Note also that such condition should not rely on the result object (i.e. the #result variable) as these are validated upfront to confirm the exclusion.

8.3.3. @CacheEvict annotation

The cache abstraction allows not just population of a cache store but also eviction. This process is useful for removing stale or unused data from the cache. Opposed to @Cacheable, annotation @CacheEvict demarcates methods that perform cache eviction, that is methods that act as triggers for removing data from the cache. Just like its sibling, @CacheEvict requires specifying one (or multiple) caches that are affected by the action, allows a custom cache and key resolution or a condition to be specified but in addition, features an extra parameter allEntries which indicates whether a cache-wide eviction needs to be performed rather then just an entry one (based on the key):

@CacheEvict(cacheNames="books", allEntries=true)

public void loadBooks(InputStream batch)

This option comes in handy when an entire cache region needs to be cleared out - rather then evicting each entry (which would take a long time since it is inefficient), all the entries are removed in one operation as shown above. Note that the framework will ignore any key specified in this scenario as it does not apply (the entire cache is evicted not just one entry).

One can also indicate whether the eviction should occur after (the default) or before the method executes through the beforeInvocation attribute. The former provides the same semantics as the rest of the annotations - once the method completes successfully, an action (in this case eviction) on the cache is executed. If the method does not execute (as it might be cached) or an exception is thrown, the eviction does not occur. The latter ( beforeInvocation=true) causes the eviction to occur always, before the method is invoked - this is useful in cases where the eviction does not need to be tied to the method outcome.

It is important to note that void methods can be used with @CacheEvict - as the methods act as triggers, the return values are ignored (as they don’t interact with the cache) - this is not the case with @Cacheable which adds/updates data into the cache and thus requires a result.

8.3.4. @Caching annotation

There are cases when multiple annotations of the same type, such as @CacheEvict or @CachePut need to be specified, for example because the condition or the key expression is different between different caches. @Caching allows multiple nested @Cacheable, @CachePut and @CacheEvict to be used on the same method:

@Caching(evict = { @CacheEvict("primary"), @CacheEvict(cacheNames="secondary", key="#p0") })

public Book importBooks(String deposit, Date date)

8.3.5. @CacheConfig annotation

So far we have seen that caching operations offered many customization options and these can be set on an operation basis. However, some of the customization options can be tedious to configure if they apply to all operations of the class. For instance, specifying the name of the cache to use for every cache operation of the class could be replaced by a single class-level definition. This is where @CacheConfig comes into play.

@CacheConfig("books")

public class BookRepositoryImpl implements BookRepository {

@Cacheable

public Book findBook(ISBN isbn) {...}

}

@CacheConfig is a class-level annotation that allows to share the cache names, the custom KeyGenerator, the custom CacheManager and finally the custom CacheResolver. Placing this annotation on the class does not turn on any caching operation.

An operation-level customization will always override a customization set on @CacheConfig. This gives therefore three levels of customizations per cache operation:

Globally configured, available for CacheManager, KeyGenerator

At class level, using @CacheConfig

At the operation level

8.3.6. Enable caching annotations

It is important to note that even though declaring the cache annotations does not automatically trigger their actions - like many things in Spring, the feature has to be declaratively enabled (which means if you ever suspect caching is to blame, you can disable it by removing only one configuration line rather than all the annotations in your code).

To enable caching annotations add the annotation @EnableCaching to one of your @Configuration classes:

@Configuration

@EnableCaching

public class AppConfig {

}

Alternatively for XML configuration use the cache:annotation-driven element:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:cache="http://www.springframework.org/schema/cache"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/cache http://www.springframework.org/schema/cache/spring-cache.xsd">

<cache:annotation-driven/>

</beans>

Both the cache:annotation-driven element and @EnableCaching annotation allow various options to be specified that influence the way the caching behavior is added to the application through AOP. The configuration is intentionally similar with that of @Transactional:

The default advice mode for processing caching annotations is "proxy" which allows for interception of calls through the proxy only; local calls within the same class cannot get intercepted that way. For a more advanced mode of interception, consider switching to "aspectj" mode in combination with compile-time or load-time weaving.

Advanced customizations using Java config require to implement CachingConfigurer: Please refer to the javadoc for more details.

Table 12. Cache annotation settings

XML Attribute Annotation Attribute Default Description

cache-manager

N/A (See CachingConfigurer javadocs)

cacheManager

Name of cache manager to use. A default CacheResolver will be initialized behind the scenes with this cache manager (or `cacheManager`if not set). For more fine-grained management of the cache resolution, consider setting the 'cache-resolver' attribute.

cache-resolver

N/A (See CachingConfigurer javadocs)

A SimpleCacheResolver using the configured cacheManager.

The bean name of the CacheResolver that is to be used to resolve the backing caches. This attribute is not required, and only needs to be specified as an alternative to the 'cache-manager' attribute.

key-generator

N/A (See CachingConfigurer javadocs)

SimpleKeyGenerator

Name of the custom key generator to use.

error-handler

N/A (See CachingConfigurer javadocs)

SimpleCacheErrorHandler

Name of the custom cache error handler to use. By default, any exception throw during a cache related operations are thrown back at the client.

mode

mode

proxy

The default mode "proxy" processes annotated beans to be proxied using Spring’s AOP framework (following proxy semantics, as discussed above, applying to method calls coming in through the proxy only). The alternative mode "aspectj" instead weaves the affected classes with Spring’s AspectJ caching aspect, modifying the target class byte code to apply to any kind of method call. AspectJ weaving requires spring-aspects.jar in the classpath as well as load-time weaving (or compile-time weaving) enabled. (See Spring configuration for details on how to set up load-time weaving.)

proxy-target-class

proxyTargetClass

false

Applies to proxy mode only. Controls what type of caching proxies are created for classes annotated with the @Cacheable or @CacheEvict annotations. If the proxy-target-class attribute is set to true, then class-based proxies are created. If proxy-target-class is false or if the attribute is omitted, then standard JDK interface-based proxies are created. (See Proxying mechanisms for a detailed examination of the different proxy types.)

order

order

Ordered.LOWEST\_PRECEDENCE

Defines the order of the cache advice that is applied to beans annotated with @Cacheable or @CacheEvict. (For more information about the rules related to ordering of AOP advice, see Advice ordering.) No specified ordering means that the AOP subsystem determines the order of the advice.

<cache:annotation-driven/> only looks for @Cacheable/@CachePut/@CacheEvict/@Caching on beans in the same application context it is defined in. This means that, if you put <cache:annotation-driven/> in a WebApplicationContext for a DispatcherServlet, it only checks for beans in your controllers, and not your services. See the MVC section for more information.

Method visibility and cache annotations

When using proxies, you should apply the cache annotations only to methods with public visibility. If you do annotate protected, private or package-visible methods with these annotations, no error is raised, but the annotated method does not exhibit the configured caching settings. Consider the use of AspectJ (see below) if you need to annotate non-public methods as it changes the bytecode itself.

Spring recommends that you only annotate concrete classes (and methods of concrete classes) with the @Cache\* annotation, as opposed to annotating interfaces. You certainly can place the @Cache\* annotation on an interface (or an interface method), but this works only as you would expect it to if you are using interface-based proxies. The fact that Java annotations are not inherited from interfaces means that if you are using class-based proxies ( proxy-target-class="true") or the weaving-based aspect ( mode="aspectj"), then the caching settings are not recognized by the proxying and weaving infrastructure, and the object will not be wrapped in a caching proxy, which would be decidedly bad.

In proxy mode (which is the default), only external method calls coming in through the proxy are intercepted. This means that self-invocation, in effect, a method within the target object calling another method of the target object, will not lead to an actual caching at runtime even if the invoked method is marked with @Cacheable - considering using the aspectj mode in this case. Also, the proxy must be fully initialized to provide the expected behaviour so you should not rely on this feature in your initialization code, i.e. @PostConstruct.

8.3.7. Using custom annotations

Custom annotation and AspectJ

This feature only works out-of-the-box with the proxy-based approach but can be enabled with a bit of extra effort using AspectJ.

The spring-aspects module defines an aspect for the standard annotations only. If you have defined your own annotations, you also need to define an aspect for those. Check AnnotationCacheAspect for an example.

The caching abstraction allows you to use your own annotations to identify what method triggers cache population or eviction. This is quite handy as a template mechanism as it eliminates the need to duplicate cache annotation declarations (especially useful if the key or condition are specified) or if the foreign imports (org.springframework) are not allowed in your code base. Similar to the rest of the stereotype annotations, @Cacheable, @CachePut, @CacheEvict and @CacheConfig can be used as meta-annotations, that is annotations that can annotate other annotations. To wit, let us replace a common @Cacheable declaration with our own, custom annotation:

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.METHOD})

@Cacheable(cacheNames="books", key="#isbn")

public @interface SlowService {

}

Above, we have defined our own SlowService annotation which itself is annotated with @Cacheable - now we can replace the following code:

@Cacheable(cacheNames="books", key="#isbn")

public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

with:

@SlowService

public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

Even though @SlowService is not a Spring annotation, the container automatically picks up its declaration at runtime and understands its meaning. Note that as mentioned above, the annotation-driven behavior needs to be enabled.

8.4. JCache (JSR-107) annotations

Since the Spring Framework 4.1, the caching abstraction fully supports the JCache standard annotations: these are @CacheResult, @CachePut, @CacheRemove and @CacheRemoveAll as well as the @CacheDefaults, @CacheKey and @CacheValue companions. These annotations can be used right the way without migrating your cache store to JSR-107: the internal implementation uses Spring’s caching abstraction and provides default CacheResolver and KeyGenerator implementations that are compliant with the specification. In other words, if you are already using Spring’s caching abstraction, you can switch to these standard annotations without changing your cache storage (or configuration, for that matter).

8.4.1. Feature summary

For those who are familiar with Spring’s caching annotations, the following table describes the main differences between the Spring annotations and the JSR-107 counterpart:

Table 13. Spring vs. JSR-107 caching annotations

Spring JSR-107 Remark

@Cacheable

@CacheResult

Fairly similar. @CacheResult can cache specific exceptions and force the execution of the method regardless of the content of the cache.

@CachePut

@CachePut

While Spring updates the cache with the result of the method invocation, JCache requires to pass it as an argument that is annotated with @CacheValue. Due to this difference, JCache allows to update the cache before or after the actual method invocation.

@CacheEvict

@CacheRemove

Fairly similar. @CacheRemove supports a conditional evict in case the method invocation results in an exception.

@CacheEvict(allEntries=true)

@CacheRemoveAll

See @CacheRemove.

@CacheConfig

@CacheDefaults

Allows to configure the same concepts, in a similar fashion.

JCache has the notion of javax.cache.annotation.CacheResolver that is identical to the Spring’s CacheResolver interface, except that JCache only supports a single cache. By default, a simple implementation retrieves the cache to use based on the name declared on the annotation. It should be noted that if no cache name is specified on the annotation, a default is automatically generated, check the javadoc of @CacheResult#cacheName() for more information.

CacheResolver instances are retrieved by a CacheResolverFactory. It is possible to customize the factory per cache operation:

@CacheResult(cacheNames="books", cacheResolverFactory=MyCacheResolverFactory.class)

public Book findBook(ISBN isbn)

For all referenced classes, Spring tries to locate a bean with the given type. If more than one match exists, a new instance is created and can use the regular bean lifecycle callbacks such as dependency injection.

Keys are generated by a javax.cache.annotation.CacheKeyGenerator that serves the same purpose as Spring’s KeyGenerator. By default, all method arguments are taken into account unless at least one parameter is annotated with @CacheKey. This is similar to Spring’s custom key generation declaration. For instance these are identical operations, one using Spring’s abstraction and the other with JCache:

@Cacheable(cacheNames="books", key="#isbn")

public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@CacheResult(cacheName="books")

public Book findBook(@CacheKey ISBN isbn, boolean checkWarehouse, boolean includeUsed)

The CacheKeyResolver to use can also be specified on the operation, in a similar fashion as the CacheResolverFactory.

JCache can manage exceptions thrown by annotated methods: this can prevent an update of the cache but it can also cache the exception as an indicator of the failure instead of calling the method again. Let’s assume that InvalidIsbnNotFoundException is thrown if the structure of the ISBN is invalid. This is a permanent failure, no book could ever be retrieved with such parameter. The following caches the exception so that further calls with the same, invalid ISBN, throws the cached exception directly instead of invoking the method again.

@CacheResult(cacheName="books", exceptionCacheName="failures"

cachedExceptions = InvalidIsbnNotFoundException.class)

public Book findBook(ISBN isbn)

8.4.2. Enabling JSR-107 support

Nothing specific needs to be done to enable the JSR-107 support alongside Spring’s declarative annotation support. Both @EnableCaching and the cache:annotation-driven element will enable automatically the JCache support if both the JSR-107 API and the spring-context-support module are present in the classpath.

Depending of your use case, the choice is basically yours. You can even mix and match services using the JSR-107 API and others using Spring’s own annotations. Be aware however that if these services are impacting the same caches, a consistent and identical key generation implementation should be used.

8.5. Declarative XML-based caching

If annotations are not an option (no access to the sources or no external code), one can use XML for declarative caching. So instead of annotating the methods for caching, one specifies the target method and the caching directives externally (similar to the declarative transaction management advice). The previous example can be translated into:

<!-- the service we want to make cacheable -->

<bean id="bookService" class="x.y.service.DefaultBookService"/>

<!-- cache definitions -->

<cache:advice id="cacheAdvice" cache-manager="cacheManager">

<cache:caching cache="books">

<cache:cacheable method="findBook" key="#isbn"/>

<cache:cache-evict method="loadBooks" all-entries="true"/>

</cache:caching>

</cache:advice>

<!-- apply the cacheable behavior to all BookService interfaces -->

<aop:config>

<aop:advisor advice-ref="cacheAdvice" pointcut="execution(\* x.y.BookService.\*(..))"/>

</aop:config>

<!-- cache manager definition omitted -->

In the configuration above, the bookService is made cacheable. The caching semantics to apply are encapsulated in the cache:advice definition which instructs method findBooks to be used for putting data into the cache while method loadBooks for evicting data. Both definitions are working against the books cache.

The aop:config definition applies the cache advice to the appropriate points in the program by using the AspectJ pointcut expression (more information is available in Aspect Oriented Programming with Spring). In the example above, all methods from the BookService are considered and the cache advice applied to them.

The declarative XML caching supports all of the annotation-based model so moving between the two should be fairly easy - further more both can be used inside the same application. The XML based approach does not touch the target code however it is inherently more verbose; when dealing with classes with overloaded methods that are targeted for caching, identifying the proper methods does take an extra effort since the method argument is not a good discriminator - in these cases, the AspectJ pointcut can be used to cherry pick the target methods and apply the appropriate caching functionality. However through XML, it is easier to apply a package/group/interface-wide caching (again due to the AspectJ pointcut) and to create template-like definitions (as we did in the example above by defining the target cache through the cache:definitions cache attribute).

8.6. Configuring the cache storage

Out of the box, the cache abstraction provides several storage integration. To use them, one needs to simply declare an appropriate CacheManager - an entity that controls and manages Caches and can be used to retrieve these for storage.

8.6.1. JDK ConcurrentMap-based Cache

The JDK-based Cache implementation resides under org.springframework.cache.concurrent package. It allows one to use ConcurrentHashMap as a backing Cache store.

<!-- simple cache manager -->

<bean id="cacheManager" class="org.springframework.cache.support.SimpleCacheManager">

<property name="caches">

<set>

<bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:name="default"/>

<bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:name="books"/>

</set>

</property>

</bean>

The snippet above uses the SimpleCacheManager to create a CacheManager for the two nested ConcurrentMapCache instances named default and books. Note that the names are configured directly for each cache.

As the cache is created by the application, it is bound to its lifecycle, making it suitable for basic use cases, tests or simple applications. The cache scales well and is very fast but it does not provide any management or persistence capabilities nor eviction contracts.

8.6.2. Ehcache-based Cache

Ehcache 3.x is fully JSR-107 compliant and no dedicated support is required for it.

The Ehcache 2.x implementation is located under org.springframework.cache.ehcache package. Again, to use it, one simply needs to declare the appropriate CacheManager:

<bean id="cacheManager"

class="org.springframework.cache.ehcache.EhCacheCacheManager" p:cache-manager-ref="ehcache"/>

<!-- EhCache library setup -->

<bean id="ehcache"

class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean" p:config-location="ehcache.xml"/>

This setup bootstraps the ehcache library inside Spring IoC (through the ehcache bean) which is then wired into the dedicated CacheManager implementation. Note the entire ehcache-specific configuration is read from ehcache.xml.

8.6.3. Caffeine Cache

Caffeine is a Java 8 rewrite of Guava’s cache and its implementation is located under org.springframework.cache.caffeine package and provides access to several features of Caffeine.

Configuring a CacheManager that creates the cache on demand is straightforward:

<bean id="cacheManager"

class="org.springframework.cache.caffeine.CaffeineCacheManager"/>

It is also possible to provide the caches to use explicitly. In that case, only those will be made available by the manager:

<bean id="cacheManager" class="org.springframework.cache.caffeine.CaffeineCacheManager">

<property name="caches">

<set>

<value>default</value>

<value>books</value>

</set>

</property>

</bean>

The Caffeine CacheManager also supports customs Caffeine and CacheLoader. See the Caffeine documentation for more information about those.

8.6.4. GemFire-based Cache

GemFire is a memory-oriented/disk-backed, elastically scalable, continuously available, active (with built-in pattern-based subscription notifications), globally replicated database and provides fully-featured edge caching. For further information on how to use GemFire as a CacheManager (and more), please refer to the Spring Data GemFire reference documentation.

8.6.5. JSR-107 Cache

JSR-107 compliant caches can also be used by Spring’s caching abstraction. The JCache implementation is located under org.springframework.cache.jcache package.

Again, to use it, one simply needs to declare the appropriate CacheManager:

<bean id="cacheManager"

class="org.springframework.cache.jcache.JCacheCacheManager"

p:cache-manager-ref="jCacheManager"/>

<!-- JSR-107 cache manager setup -->

<bean id="jCacheManager" .../>

8.6.6. Dealing with caches without a backing store

Sometimes when switching environments or doing testing, one might have cache declarations without an actual backing cache configured. As this is an invalid configuration, at runtime an exception will be thrown since the caching infrastructure is unable to find a suitable store. In situations like this, rather then removing the cache declarations (which can prove tedious), one can wire in a simple, dummy cache that performs no caching - that is, forces the cached methods to be executed every time:

<bean id="cacheManager" class="org.springframework.cache.support.CompositeCacheManager">

<property name="cacheManagers">

<list>

<ref bean="jdkCache"/>

<ref bean="gemfireCache"/>

</list>

</property>

<property name="fallbackToNoOpCache" value="true"/>

</bean>

The CompositeCacheManager above chains multiple CacheManagers and additionally, through the fallbackToNoOpCache flag, adds a no op cache that for all the definitions not handled by the configured cache managers. That is, every cache definition not found in either jdkCache or gemfireCache (configured above) will be handled by the no op cache, which will not store any information causing the target method to be executed every time.

8.7. Plugging-in different back-end caches

Clearly there are plenty of caching products out there that can be used as a backing store. To plug them in, one needs to provide a CacheManager and Cache implementation since unfortunately there is no available standard that we can use instead. This may sound harder than it is since in practice, the classes tend to be simple adapters that map the caching abstraction framework on top of the storage API as the ehcache classes can show. Most CacheManager classes can use the classes in org.springframework.cache.support package, such as AbstractCacheManager which takes care of the boiler-plate code leaving only the actual mapping to be completed. We hope that in time, the libraries that provide integration with Spring can fill in this small configuration gap.

8.8. How can I set the TTL/TTI/Eviction policy/XXX feature?

Directly through your cache provider. The cache abstraction is…​ well, an abstraction not a cache implementation. The solution you are using might support various data policies and different topologies which other solutions do not (take for example the JDK ConcurrentHashMap) - exposing that in the cache abstraction would be useless simply because there would no backing support. Such functionality should be controlled directly through the backing cache, when configuring it or through its native API.

9. Appendix

9.1. XML Schemas

This part of the appendix lists XML schemas related to integration technologies.

9.1.1. The jee schema

The jee tags deal with Java EE (Java Enterprise Edition)-related configuration issues, such as looking up a JNDI object and defining EJB references.

To use the tags in the jee schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the jee namespace are available to you.

<?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:jee="http://www.springframework.org/schema/jee" xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee.xsd"> <!-- bean definitions here -->

</beans>

<jee:jndi-lookup/> (simple)

Before…​

<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName" value="jdbc/MyDataSource"/>

</bean>

<bean id="userDao" class="com.foo.JdbcUserDao">

<!-- Spring will do the cast automatically (as usual) -->

<property name="dataSource" ref="dataSource"/>

</bean>

After…​

<jee:jndi-lookup id="dataSource" jndi-name="jdbc/MyDataSource"/>

<bean id="userDao" class="com.foo.JdbcUserDao">

<!-- Spring will do the cast automatically (as usual) -->

<property name="dataSource" ref="dataSource"/>

</bean>

<jee:jndi-lookup/> (with single JNDI environment setting)

Before…​

<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName" value="jdbc/MyDataSource"/>

<property name="jndiEnvironment">

<props>

<prop key="foo">bar</prop>

</props>

</property>

</bean>

After…​

<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">

<jee:environment>foo=bar</jee:environment>

</jee:jndi-lookup>

<jee:jndi-lookup/> (with multiple JNDI environment settings)

Before…​

<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName" value="jdbc/MyDataSource"/>

<property name="jndiEnvironment">

<props>

<prop key="foo">bar</prop>

<prop key="ping">pong</prop>

</props>

</property>

</bean>

After…​

<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">

<!-- newline-separated, key-value pairs for the environment (standard Properties format) -->

<jee:environment>

foo=bar

ping=pong

</jee:environment>

</jee:jndi-lookup>

<jee:jndi-lookup/> (complex)

Before…​

<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName" value="jdbc/MyDataSource"/>

<property name="cache" value="true"/>

<property name="resourceRef" value="true"/>

<property name="lookupOnStartup" value="false"/>

<property name="expectedType" value="com.myapp.DefaultFoo"/>

<property name="proxyInterface" value="com.myapp.Foo"/>

</bean>

After…​

<jee:jndi-lookup id="simple"

jndi-name="jdbc/MyDataSource"

cache="true"

resource-ref="true"

lookup-on-startup="false"

expected-type="com.myapp.DefaultFoo"

proxy-interface="com.myapp.Foo"/>

<jee:local-slsb/> (simple)

The <jee:local-slsb/> tag configures a reference to an EJB Stateless SessionBean.

Before…​

<bean id="simple"

class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">

<property name="jndiName" value="ejb/RentalServiceBean"/>

<property name="businessInterface" value="com.foo.service.RentalService"/>

</bean>

After…​

<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"

business-interface="com.foo.service.RentalService"/>

<jee:local-slsb/> (complex)

<bean id="complexLocalEjb"

class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">

<property name="jndiName" value="ejb/RentalServiceBean"/>

<property name="businessInterface" value="com.foo.service.RentalService"/>

<property name="cacheHome" value="true"/>

<property name="lookupHomeOnStartup" value="true"/>

<property name="resourceRef" value="true"/>

</bean>

After…​

<jee:local-slsb id="complexLocalEjb"

jndi-name="ejb/RentalServiceBean"

business-interface="com.foo.service.RentalService"

cache-home="true"

lookup-home-on-startup="true"

resource-ref="true">

<jee:remote-slsb/>

The <jee:remote-slsb/> tag configures a reference to a remote EJB Stateless SessionBean.

Before…​

<bean id="complexRemoteEjb"

class="org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean">

<property name="jndiName" value="ejb/MyRemoteBean"/>

<property name="businessInterface" value="com.foo.service.RentalService"/>

<property name="cacheHome" value="true"/>

<property name="lookupHomeOnStartup" value="true"/>

<property name="resourceRef" value="true"/>

<property name="homeInterface" value="com.foo.service.RentalService"/>

<property name="refreshHomeOnConnectFailure" value="true"/>

</bean>

After…​

<jee:remote-slsb id="complexRemoteEjb"

jndi-name="ejb/MyRemoteBean"

business-interface="com.foo.service.RentalService"

cache-home="true"

lookup-home-on-startup="true"

resource-ref="true"

home-interface="com.foo.service.RentalService"

refresh-home-on-connect-failure="true">

9.1.2. The jms schema

The jms tags deal with configuring JMS-related beans such as Spring’s MessageListenerContainers. These tags are detailed in the section of the JMS chapter entitled JMS namespace support. Please do consult that chapter for full details on this support and the jms tags themselves.

In the interest of completeness, to use the tags in the jms schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the jms namespace are available to you.

<?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:jms="http://www.springframework.org/schema/jms" xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms.xsd"> <!-- bean definitions here -->

</beans>

9.1.3. <context:mbean-export/>

This element is detailed in Configuring annotation based MBean export.

9.1.4. The cache schema

The cache tags can be used to enable support for Spring’s @CacheEvict, @CachePut and @Caching annotations. It it also supports declarative XML-based caching. See Enable caching annotations and Declarative XML-based caching for details.

To use the tags in the cache schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the cache namespace are available to you.

<?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:cache="http://www.springframework.org/schema/cache" xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/cache http://www.springframework.org/schema/cache/spring-cache.xsd"> <!-- bean definitions here -->

</beans>

## Language Support

Version 5.0.8.RELEASE

### 1. Kotlin

Kotlin is a statically-typed language targeting the JVM (and other platforms) which allows writing concise and elegant code while providing very good interoperability with existing libraries written in Java.

The Spring Framework provides first-class support for Kotlin that allows developers to write Kotlin applications almost as if the Spring Framework was a native Kotlin framework.

The easiest way to learn about Spring + Kotlin is to follow this comprehensive tutorial. Feel free to join the #spring channel of Kotlin Slack or ask a question with spring and kotlin tags on Stackoverflow if you need support.

#### 1.1. Requirements

Spring Framework supports Kotlin 1.1+ and requires kotlin-stdlib (or one of its variants like kotlin-stdlib-jre8 for Kotlin 1.1 or kotlin-stdlib-jdk8 for Kotlin 1.2+) and kotlin-reflect to be present on the classpath. They are provided by default if one bootstraps a Kotlin project on start.spring.io.

#### 1.2. Extensions

Kotlin extensions provide the ability to extend existing classes with additional functionality. The Spring Framework Kotlin APIs make use of these extensions to add new Kotlin specific conveniences to existing Spring APIs.

Spring Framework KDoc API lists and documents all the Kotlin extensions and DSLs available.

|  |
| --- |
| *Keep in mind that Kotlin extensions need to be imported to be used. This means for example that the GenericApplicationContext.registerBean Kotlin extension will only be available if import org.springframework.context.support.registerBean is imported. That said, similar to static imports, an IDE should automatically suggest the import in most cases.* |

Keep in mind that Kotlin extensions need to be imported to be used. This means for example that the GenericApplicationContext.registerBean Kotlin extension will only be available if import org.springframework.context.support.registerBean is imported. That said, similar to static imports, an IDE should automatically suggest the import in most cases.

For example, Kotlin reified type parameters provide a workaround for JVM generics type erasure, and Spring Framework provides some extensions to take advantage of this feature. This allows for a better Kotlin API RestTemplate, the new WebClient from Spring WebFlux and for various other APIs.

|  |
| --- |
| *Other libraries like Reactor and Spring Data also provide Kotlin extensions for their APIs, thus giving a better Kotlin development experience overall.* |

To retrieve a list of Foo objects in Java, one would normally write:

|  |
| --- |
| Flux<User> users = client.get().retrieve().bodyToFlux(User.class) |

Whilst with Kotlin and Spring Framework extensions, one is able to write:

|  |
| --- |
| val users = client.get().retrieve().bodyToFlux<User>()  // or (both are equivalent)  val users : Flux<User> = client.get().retrieve().bodyToFlux()aaaa |

As in Java, users in Kotlin is strongly typed, but Kotlin’s clever type inference allows for shorter syntax.

#### 1.3. Null-safety

One of Kotlin’s key features is null-safety - which cleanly deals with null values at compile time rather than bumping into the famous NullPointerException at runtime. This makes applications safer through nullability declarations and expressing "value or no value" semantics without paying the cost of wrappers like Optional. (Kotlin allows using functional constructs with nullable values; check out this comprehensive guide to Kotlin null-safety.)

Although Java does not allow one to express null-safety in its type-system, Spring Framework now provides null-safety of the whole Spring Framework API via tooling-friendly annotations declared in the org.springframework.lang package. By default, types from Java APIs used in Kotlin are recognized as platform types for which null-checks are relaxed. Kotlin support for JSR 305 annotations + Spring nullability annotations provide null-safety for the whole Spring Framework API to Kotlin developers, with the advantage of dealing with null related issues at compile time.

|  |
| --- |
| *Libraries like Reactor or Spring Data provide null-safe APIs leveraging this feature.* |

The JSR 305 checks can be configured by adding the -Xjsr305 compiler flag with the following options: -Xjsr305={strict|warn|ignore}.

For kotlin versions 1.1+, the default behavior is the same to -Xjsr305=warn. The strict value is required to have Spring Framework API null-safety taken in account in Kotlin types inferred from Spring API but should be used with the knowledge that Spring API nullability declaration could evolve even between minor releases and more checks may be added in the future).

|  |
| --- |
| *Generic type arguments, varargs and array elements nullability are not supported yet, but should be in an upcoming release, see this discussion for up-to-date information.* |

#### 1.4. Classes & Interfaces

Spring Framework supports various Kotlin constructs like instantiating Kotlin classes via primary constructors, immutable classes data binding and function optional parameters with default values.

Kotlin parameter names are recognized via a dedicated KotlinReflectionParameterNameDiscoverer which allows finding interface method parameter names without requiring the Java 8 -parameters compiler flag enabled during compilation.

Jackson Kotlin module which is required for serializing / deserializing JSON data is automatically registered when found in the classpath and a warning message will be logged if Jackson and Kotlin are detected without the Jackson Kotlin module present.

#### 1.5. Annotations

Spring Framework also takes advantage of Kotlin null-safety to determine if a HTTP parameter is required without having to explicitly define the required attribute. That means @RequestParam name: String? will be treated as not required and conversely @RequestParam name: String as being required. This feature is also supported on the Spring Messaging @Header annotation.

In a similar fashion, Spring bean injection with @Autowired, @Bean or @Inject uses this information to determine if a bean is required or not.

For example, @Autowired lateinit var foo: Foo implies that a bean of type Foo must be registered in the application context, while @Autowired lateinit var foo: Foo? won’t raise an error if such bean does not exist.

Following the same principle, @Bean fun baz(foo: Foo, bar: Bar?) = Baz(foo, bar) implies that a bean of type Foo must be registered in the application context while a bean of type Bar may or may not exist. The same behavior applies to autowired constructor parameters.

|  |
| --- |
| *If you are using bean validation on classes with properties or a primary constructor parameters, you may need to leverage annotation use-site targets like @field:NotNull or @get:Size(min=5, max=15) as described in this Stack Overflow response.* |

#### 1.6. Bean definition DSL

Spring Framework 5 introduces a new way to register beans in a functional way using lambdas as an alternative to XML or JavaConfig (@Configuration and @Bean). In a nutshell, it makes it possible to register beans with a lambda that acts as a FactoryBean. This mechanism is very efficient as it does not require any reflection or CGLIB proxies.

In Java, one may for example write:

|  |
| --- |
| GenericApplicationContext context = new GenericApplicationContext();  context.registerBean(Foo.class);  context.registerBean(Bar.class, () -> new Bar(context.getBean(Foo.class))  ); |

Whilst in Kotlin with reified type parameters and GenericApplicationContext Kotlin extensions one can instead simply write:

|  |
| --- |
| val context = GenericApplicationContext().apply {  registerBean<Foo>()  registerBean { Bar(it.getBean<Foo>()) }  } |

In order to allow a more declarative approach and cleaner syntax, Spring Framework provides a Kotlin bean definition DSL It declares an ApplicationContextInitializer via a clean declarative API which enables one to deal with profiles and Environment for customizing how beans are registered.

|  |
| --- |
| fun beans() = beans {  bean<UserHandler>()  bean<Routes>()  bean<WebHandler>("webHandler") {  RouterFunctions.toWebHandler(  ref<Routes>().router(),  HandlerStrategies.builder().viewResolver(ref()).build()  )  }  bean("messageSource") {  ReloadableResourceBundleMessageSource().apply {  setBasename("messages")  setDefaultEncoding("UTF-8")  }  }  bean {  val prefix = "classpath:/templates/"  val suffix = ".mustache"  val loader = MustacheResourceTemplateLoader(prefix, suffix)  MustacheViewResolver(Mustache.compiler().withLoader(loader)).apply {  setPrefix(prefix)  setSuffix(suffix)  }  }  profile("foo") {  bean<Foo>()  }  } |

In this example, bean<Routes>() is using autowiring by constructor and ref<Routes>() is a shortcut for applicationContext.getBean(Routes::class.java).

This beans() function can then be used to register beans on the application context.

|  |
| --- |
| val context = GenericApplicationContext().apply {  beans().initialize(this)  refresh()  } |

|  |
| --- |
| *This DSL is programmatic, thus it allows custom registration logic of beans via an if expression, a for loop or any other Kotlin constructs.* |

See spring-kotlin-functional beans declaration for a concrete example.

|  |
| --- |
| *Spring Boot is based on Java Config and does not provide specific support for functional bean definition yet, but one can experimentally use functional bean definitions via Spring Boot’s ApplicationContextInitializer support, see this Stack Overflow answer for more details and up-to-date information.* |

#### 1.7. Web

##### 1.7.1. WebFlux Functional DSL

Spring Framework now comes with a Kotlin routing DSL that allows one to leverage the WebFlux functional API for writing clean and idiomatic Kotlin code:

|  |
| --- |
| router {  accept(TEXT\_HTML).nest {  GET("/") { ok().render("index") }  GET("/sse") { ok().render("sse") }  GET("/users", userHandler::findAllView)  }  "/api".nest {  accept(APPLICATION\_JSON).nest {  GET("/users", userHandler::findAll)  }  accept(TEXT\_EVENT\_STREAM).nest {  GET("/users", userHandler::stream)  }  }  resources("/\*\*", ClassPathResource("static/"))  } |

|  |
| --- |
| *This DSL is programmatic, thus it allows custom registration logic of beans via an if expression, a for loop or any other Kotlin constructs. That can be useful when routes need to be registered depending on dynamic data (for example, from a database).* |

See MiXiT project routes for a concrete example.

##### 1.7.2. Kotlin Script templates

As of version 4.3, Spring Framework provides a ScriptTemplateView to render templates using script engines that supports JSR-223. Spring Framework 5 goes even further by extending this feature to WebFlux and supporting i18n and nested templates.

Kotlin provides similar support and allows the rendering of Kotlin based templates, see this commit for details.

This enables some interesting use cases - like writing type-safe templates using kotlinx.html DSL or simply using Kotlin multiline String with interpolation.

This can allow one to write Kotlin templates with full autocompletion and refactoring support in a supported IDE:

|  |
| --- |
| import io.spring.demo.\*  """  ${include("header")}  <h1>${i18n("title")}</h1>  <ul>  ${users.joinToLine{ "<li>${i18n("user")} ${it.firstname} ${it.lastname}</li>" }}  </ul>  ${include("footer")}  """ |

See kotlin-script-templating example project for more details.

#### 1.8. Spring projects in Kotlin

This section provides focus on some specific hints and recommendations worth knowing when developing Spring projects in Kotlin.

##### 1.8.1. Final by default

By default, all classes in Kotlin are final. The open modifier on a class is the opposite of Java’s final: it allows others to inherit from this class. This also applies to member functions, in that they need to be marked as open to be overridden.

Whilst Kotlin’s JVM-friendly design is generally frictionless with Spring, this specific Kotlin feature can prevent the application from starting, if this fact is not taken in consideration. This is because Spring beans are normally proxied by CGLIB - such as @Configuration classes - which need to be inherited at runtime for technical reasons. The workaround was to add an open keyword on each class and member functions of Spring beans proxied by CGLIB such as @Configuration classes, which can quickly become painful and is against the Kotlin principle of keeping code concise and predictable.

Fortunately, Kotlin now provides a kotlin-spring plugin, a preconfigured version of kotlin-allopen plugin that automatically opens classes and their member functions for types annotated or meta-annotated with one of the following annotations:

@Component

@Async

@Transactional

@Cacheable

Meta-annotations support means that types annotated with @Configuration, @Controller, @RestController, @Service or @Repository are automatically opened since these annotations are meta-annotated with @Component.

start.spring.io enables it by default, so in practice you will be able to write your Kotlin beans without any additional open keyword, like in Java.

##### 1.8.2. Using immutable class instances for persistence

In Kotlin, it is very convenient and considered best practice to declare read-only properties within the primary constructor, as in the following example:

|  |
| --- |
| class Person(val name: String, val age: Int) |

You can optionally add the data keyword to make the compiler automatically derive the following members from all properties declared in the primary constructor:

equals()/hashCode() pair

toString() of the form "User(name=John, age=42)"

componentN() functions corresponding to the properties in their order of declaration

copy() function

This allows for easy changes to individual properties even if Person properties are read-only:

|  |
| --- |
| data class Person(val name: String, val age: Int)  val jack = Person(name = "Jack", age = 1)  val olderJack = jack.copy(age = 2) |

Common persistence technologies such as JPA require a default constructor, preventing this kind of design. Fortunately, there is now a workaround for this "default constructor hell" since Kotlin provides a kotlin-jpa plugin which generates synthetic no-arg constructor for classes annotated with JPA annotations.

If you need to leverage this kind of mechanism for other persistence technologies, you can configure the kotlin-noarg plugin.

|  |
| --- |
| *As of the Kay release train, Spring Data supports Kotlin immutable class instances and does not require the kotlin-noarg plugin if the module leverages Spring Data object mappings (like with MongoDB, Redis, Cassandra, etc).* |

##### 1.8.3. Injecting dependencies

Our recommendation is to try and favor constructor injection with val read-only (and non-nullable when possible) properties.

|  |
| --- |
| @Component  class YourBean(  private val mongoTemplate: MongoTemplate,  private val solrClient: SolrClient  ) |

|  |
| --- |
| *As of Spring Framework 4.3, classes with a single constructor have their parameters automatically autowired, that’s why there is no need for an explicit @Autowired constructor in the example shown above.* |

If one really needs to use field injection, use the lateinit var construct, i.e.,

|  |
| --- |
| @Component  class YourBean {  @Autowired  lateinit var mongoTemplate: MongoTemplate  @Autowired  lateinit var solrClient: SolrClient  } |

##### 1.8.4. Injecting configuration properties

In Java, one can inject configuration properties using annotations like @Value("${property}"), however in Kotlin $ is a reserved character that is used for string interpolation.

Therefore, if one wishes to use the @Value annotation in Kotlin, the $ character will need to be escaped by writing @Value("\${property}").

As an alternative, it is possible to customize the properties placeholder prefix by declaring the following configuration beans:

|  |
| --- |
| @Bean  fun propertyConfigurer() = PropertySourcesPlaceholderConfigurer().apply {  setPlaceholderPrefix("%{")  } |

Existing code (like Spring Boot actuators or @LocalServerPort) that uses the ${…​} syntax, can be customised with configuration beans, like as follows:

|  |
| --- |
| @Bean  fun kotlinPropertyConfigurer() = PropertySourcesPlaceholderConfigurer().apply {  setPlaceholderPrefix("%{")  setIgnoreUnresolvablePlaceholders(true)  }  @Bean  fun defaultPropertyConfigurer() = PropertySourcesPlaceholderConfigurer() |

|  |
| --- |
| *If Spring Boot is being used, then @ConfigurationProperties instead of @Value annotations can be used, but currently this only works with lateinit or nullable var properties (the former is recommended) since immutable classes initialized by constructors are not yet supported. See these issues about @ConfigurationProperties binding for immutable POJOs and @ConfigurationProperties binding on interfaces for more details.* |

##### 1.8.5. Annotation array attributes

Kotlin annotations are mostly similar to Java ones, but array attributes - which are extensively used in Spring - behave differently. As explained in Kotlin documentation unlike other attributes, the value attribute name can be omitted and specified as a vararg parameter.

To understand what that means, let’s take @RequestMapping, which is one of the most widely used Spring annotations as an example. This Java annotation is declared as:

|  |
| --- |
| public @interface RequestMapping {  @AliasFor("path")  String[] value() default {};  @AliasFor("value")  String[] path() default {};  RequestMethod[] method() default {};  // ...  } |

The typical use case for @RequestMapping is to map a handler method to a specific path and method. In Java, it is possible to specify a single value for the annotation array attribute and it will be automatically converted to an array.

That’s why one can write @RequestMapping(value = "/foo", method = RequestMethod.GET) or @RequestMapping(path = "/foo", method = RequestMethod.GET).

However, in Kotlin 1.2+, one will have to write @RequestMapping("/foo", method = [RequestMethod.GET]) or @RequestMapping(path = ["/foo"], method = [RequestMethod.GET]) (square brackets need to be specified with named array attributes).

An alternative for this specific method attribute (the most common one) is to use a shortcut annotation such as @GetMapping or @PostMapping, etc.

|  |
| --- |
| *Reminder: If the @RequestMapping method attribute is not specified, all HTTP methods will be matched, not only the GET one.* |

##### 1.8.6. Testing

Per class lifecycle

Kotlin allows one to specify meaningful test function names between backticks, and as of JUnit 5 Kotlin test classes can use the @TestInstance(TestInstance.Lifecycle.PER\_CLASS) annotation to enable a single instantiation of test classes which allows the use of @BeforeAll and @AfterAll annotations on non-static methods, which is a good fit for Kotlin.

It is now possible to change the default behavior to PER\_CLASS thanks to a junit-platform.properties file with a junit.jupiter.testinstance.lifecycle.default = per\_class property.

|  |
| --- |
| class IntegrationTests {  val application = Application(8181)  val client = WebClient.create("http://localhost:8181")  @BeforeAll  fun beforeAll() {  application.start()  }  @Test  fun `Find all users on HTML page`() {  client.get().uri("/users")  .accept(TEXT\_HTML)  .retrieve()  .bodyToMono<String>()  .test()  .expectNextMatches { it.contains("Foo") }  .verifyComplete()  }  @AfterAll  fun afterAll() {  application.stop()  }  } |

Specification-like tests

It is possible to create specification-like tests with JUnit 5 and Kotlin.

|  |
| --- |
| class SpecificationLikeTests {  @Nested  @DisplayName("a calculator")  inner class Calculator {  val calculator = SampleCalculator()  @Test  fun `should return the result of adding the first number to the second number`() {  val sum = calculator.sum(2, 4)  assertEquals(6, sum)  }  @Test  fun `should return the result of subtracting the second number from the first number`() {  val subtract = calculator.subtract(4, 2)  assertEquals(2, subtract)  }  }  } |

WebTestClient type inference issue in Kotlin

Due to a type inference issue, make sure to use Kotlin expectBody extension (like .expectBody<String>().isEqualTo("foo")) since it provides a workaround for the Kotlin issue with the Java API.

See also the related SPR-16057 issue.

#### 1.9. Getting started

##### 1.9.1. start.spring.io

The easiest way to start a new Spring Framework 5 project in Kotlin is to create a new Spring Boot 2 project on start.spring.io.

It is also possible to create a standalone WebFlux project as described in this blog post.

##### 1.9.2. Choosing the web flavor

Spring Framework now comes with 2 different web stacks: Spring MVC and Spring WebFlux.

Spring WebFlux is recommended if one wants to create applications that will deal with latency, long-lived connections, streaming scenarios or simply if one wants to use the web functional Kotlin DSL.

For other use cases, especially if you are using blocking technologies like JPA, Spring MVC and its annotation-based programming model is a perfectly valid and fully supported choice.

#### 1.10. Resources

Kotlin language reference

Kotlin Slack (with a dedicated #spring channel)

Stackoverflow with spring and kotlin tags

Try Kotlin in your browser

Kotlin blog

Awesome Kotlin

##### 1.10.1. Tutorials

Building web applications with Spring Boot and Kotlin

Creating a RESTful Web Service with Spring Boot

##### 1.10.2. Blog posts

Developing Spring Boot applications with Kotlin

A Geospatial Messenger with Kotlin, Spring Boot and PostgreSQL

Introducing Kotlin support in Spring Framework 5.0

Spring Framework 5 Kotlin APIs, the functional way

##### 1.10.3. Examples

spring-boot-kotlin-demo: regular Spring Boot + Spring Data JPA project

mixit: Spring Boot 2 + WebFlux + Reactive Spring Data MongoDB

spring-kotlin-functional: standalone WebFlux + functional bean definition DSL

spring-kotlin-fullstack: WebFlux Kotlin fullstack example with Kotlin2js for frontend instead of JavaScript or TypeScript

spring-petclinic-kotlin: Kotlin version of the Spring PetClinic Sample Application

spring-kotlin-deepdive: a step by step migration for Boot 1.0 + Java to Boot 2.0 + Kotlin

##### 1.10.4. Issues

Here is a list of pending issues related to Spring + Kotlin support.

Spring Framework

Unable to use WebTestClient with mock server in Kotlin

Support null-safety at generics, varargs and array elements level

Add support for Kotlin coroutines

Spring Boot

Allow @ConfigurationProperties binding for immutable POJOs

Allow @ConfigurationProperties binding on interfaces

Expose the functional bean registration API via SpringApplication

Add null-safety annotations on Spring Boot APIs

Use Kotlin’s bom to provide dependency management for Kotlin

Kotlin

Parent issue for Spring Framework support

Kotlin requires type inference where Java doesn’t

Better generics null-safety support

Smart cast regression with open classes

Impossible to pass not all SAM argument as function

Apply JSR 305 meta-annotations to generic type parameters

Provide a way for libraries to avoid mixing Kotlin 1.0 and 1.1 dependencies

Support JSR 223 bindings directly via script variables

Support all-open and no-arg compiler plugins in Kotlin Eclipse plugin

### 2. Apache Groovy

Groovy is a powerful, optionally typed and dynamic language, with static-typing and static compilation capabilities. It offers a concise syntax and integrates smoothly with any existing Java application.

The Spring Framework provides a dedicated ApplicationContext that supports a Groovy-based Bean Definition DSL. For more details, see The Groovy Bean Definition DSL.

Further support for Groovy including beans written in Groovy, refreshable script beans, and more is available in next the section Dynamic Language Support.

### 3. Dynamic Language Support

#### 3.1. Introduction

Spring 2.0 introduces comprehensive support for using classes and objects that have been defined using a dynamic language (such as JRuby) with Spring. This support allows you to write any number of classes in a supported dynamic language, and have the Spring container transparently instantiate, configure and dependency inject the resulting objects.

The dynamic languages currently supported are:

JRuby 1.5+

Groovy 1.8+

BeanShell 2.0

|  |
| --- |
| *Why only these languages?*  *The supported languages were chosen because a) the languages have a lot of traction in the Java enterprise community, b) no requests were made for other languages at the time that this support was added, and c) the Spring developers were most familiar with them.* |

Fully working examples of where this dynamic language support can be immediately useful are described in Scenarios.

#### 3.2. A first example

This bulk of this chapter is concerned with describing the dynamic language support in detail. Before diving into all of the ins and outs of the dynamic language support, let’s look at a quick example of a bean defined in a dynamic language. The dynamic language for this first bean is Groovy (the basis of this example was taken from the Spring test suite, so if you want to see equivalent examples in any of the other supported languages, take a look at the source code).

Find below the Messenger interface that the Groovy bean is going to be implementing, and note that this interface is defined in plain Java. Dependent objects that are injected with a reference to the Messenger won’t know that the underlying implementation is a Groovy script.

|  |
| --- |
| package org.springframework.scripting;  public interface Messenger {  String getMessage();  } |

Here is the definition of a class that has a dependency on the Messenger interface.

|  |
| --- |
| package org.springframework.scripting;  public class DefaultBookingService implements BookingService {  private Messenger messenger;  public void setMessenger(Messenger messenger) {  this.messenger = messenger;  }  public void processBooking() {  // use the injected Messenger object...  }  } |

Here is an implementation of the Messenger interface in Groovy.

|  |
| --- |
| // from the file 'Messenger.groovy'  package org.springframework.scripting.groovy;  // import the Messenger interface (written in Java) that is to be implemented  import org.springframework.scripting.Messenger  // define the implementation in Groovy  class GroovyMessenger implements Messenger {  String message  } |

Finally, here are the bean definitions that will effect the injection of the Groovy-defined Messenger implementation into an instance of the DefaultBookingService class.

|  |
| --- |
| *To use the custom dynamic language tags to define dynamic-language-backed beans, you need to have the XML Schema preamble at the top of your Spring XML configuration file. You also need to be using a Spring ApplicationContext implementation as your IoC container. Using the dynamic-language-backed beans with a plain BeanFactory implementation is supported, but you have to manage the plumbing of the Spring internals to do so.*  *For more information on schema-based configuration, see XML Schema-based configuration.* |

|  |
| --- |
| <?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:lang="http://www.springframework.org/schema/lang"  xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang.xsd">  <!-- this is the bean definition for the Groovy-backed Messenger implementation -->  <lang:groovy id="messenger" script-source="classpath:Messenger.groovy">  <lang:property name="message" value="I Can Do The Frug" />  </lang:groovy>  <!-- an otherwise normal bean that will be injected by the Groovy-backed Messenger -->  <bean id="bookingService" class="x.y.DefaultBookingService">  <property name="messenger" ref="messenger" />  </bean>  </beans> |

The bookingService bean (a DefaultBookingService) can now use its private messenger member variable as normal because the Messenger instance that was injected into it is a Messenger instance. There is nothing special going on here, just plain Java and plain Groovy.

Hopefully the above XML snippet is self-explanatory, but don’t worry unduly if it isn’t. Keep reading for the in-depth detail on the whys and wherefores of the above configuration.

#### 3.3. Defining beans that are backed by dynamic languages

This section describes exactly how you define Spring managed beans in any of the supported dynamic languages.

Please note that this chapter does not attempt to explain the syntax and idioms of the supported dynamic languages. For example, if you want to use Groovy to write certain of the classes in your application, then the assumption is that you already know Groovy. If you need further details about the dynamic languages themselves, please consult Further Resources at the end of this chapter.

##### 3.3.1. Common concepts

The steps involved in using dynamic-language-backed beans are as follows:

Write the test for the dynamic language source code (naturally)

Then write the dynamic language source code itself :)

Define your dynamic-language-backed beans using the appropriate <lang:language/> element in the XML configuration (you can of course define such beans programmatically using the Spring API - although you will have to consult the source code for directions on how to do this as this type of advanced configuration is not covered in this chapter). Note this is an iterative step. You will need at least one bean definition per dynamic language source file (although the same dynamic language source file can of course be referenced by multiple bean definitions).

The first two steps (testing and writing your dynamic language source files) are beyond the scope of this chapter. Refer to the language specification and / or reference manual for your chosen dynamic language and crack on with developing your dynamic language source files. You will first want to read the rest of this chapter though, as Spring’s dynamic language support does make some (small) assumptions about the contents of your dynamic language source files.

The <lang:language/> element

The final step involves defining dynamic-language-backed bean definitions, one for each bean that you want to configure (this is no different from normal JavaBean configuration). However, instead of specifying the fully qualified classname of the class that is to be instantiated and configured by the container, you use the <lang:language/> element to define the dynamic language-backed bean.

Each of the supported languages has a corresponding <lang:language/> element:

<lang:groovy/> (Groovy)

<lang:bsh/> (BeanShell)

<lang:std/> (JSR-223)

The exact attributes and child elements that are available for configuration depends on exactly which language the bean has been defined in (the language-specific sections below provide the full lowdown on this).

Refreshable beans

One of the (if not the) most compelling value adds of the dynamic language support in Spring is the'refreshable bean' feature.

A refreshable bean is a dynamic-language-backed bean that with a small amount of configuration, a dynamic-language-backed bean can monitor changes in its underlying source file resource, and then reload itself when the dynamic language source file is changed (for example when a developer edits and saves changes to the file on the filesystem).

This allows a developer to deploy any number of dynamic language source files as part of an application, configure the Spring container to create beans backed by dynamic language source files (using the mechanisms described in this chapter), and then later, as requirements change or some other external factor comes into play, simply edit a dynamic language source file and have any change they make reflected in the bean that is backed by the changed dynamic language source file. There is no need to shut down a running application (or redeploy in the case of a web application). The dynamic-language-backed bean so amended will pick up the new state and logic from the changed dynamic language source file.

|  |
| --- |
| *Please note that this feature is off by default.* |

Let’s take a look at an example to see just how easy it is to start using refreshable beans. To turn on the refreshable beans feature, you simply have to specify exactly one additional attribute on the <lang:language/> element of your bean definition. So if we stick with the example from earlier in this chapter, here’s what we would change in the Spring XML configuration to effect refreshable beans:

|  |
| --- |
| <beans>  <!-- this bean is now 'refreshable' due to the presence of the 'refresh-check-delay' attribute -->  <lang:groovy id="messenger"  refresh-check-delay="5000" <!-- switches refreshing on with 5 seconds between checks -->  script-source="classpath:Messenger.groovy">  <lang:property name="message" value="I Can Do The Frug" />  </lang:groovy>  <bean id="bookingService" class="x.y.DefaultBookingService">  <property name="messenger" ref="messenger" />  </bean>  </beans> |

That really is all you have to do. The 'refresh-check-delay' attribute defined on the 'messenger' bean definition is the number of milliseconds after which the bean will be refreshed with any changes made to the underlying dynamic language source file. You can turn off the refresh behavior by assigning a negative value to the 'refresh-check-delay' attribute. Remember that, by default, the refresh behavior is disabled. If you don’t want the refresh behavior, then simply don’t define the attribute.

If we then run the following application we can exercise the refreshable feature; please do excuse the 'jumping-through-hoops-to-pause-the-execution' shenanigans in this next slice of code. The System.in.read() call is only there so that the execution of the program pauses while I (the author) go off and edit the underlying dynamic language source file so that the refresh will trigger on the dynamic-language-backed bean when the program resumes execution.

|  |
| --- |
| import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import org.springframework.scripting.Messenger;  public final class Boot {  public static void main(final String[] args) throws Exception {  ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");  Messenger messenger = (Messenger) ctx.getBean("messenger");  System.out.println(messenger.getMessage());  // pause execution while I go off and make changes to the source file...  System.in.read();  System.out.println(messenger.getMessage());  }  } |

Let’s assume then, for the purposes of this example, that all calls to the getMessage() method of Messenger implementations have to be changed such that the message is surrounded by quotes. Below are the changes that I (the author) make to the Messenger.groovy source file when the execution of the program is paused.

|  |
| --- |
| package org.springframework.scripting  class GroovyMessenger implements Messenger {  private String message = "Bingo"  public String getMessage() {  // change the implementation to surround the message in quotes  return "'" + this.message + "'"  }  public void setMessage(String message) {  this.message = message  }  } |

When the program executes, the output before the input pause will be I Can Do The Frug. After the change to the source file is made and saved, and the program resumes execution, the result of calling the getMessage() method on the dynamic-language-backed Messenger implementation will be 'I Can Do The Frug' (notice the inclusion of the additional quotes).

It is important to understand that changes to a script will not trigger a refresh if the changes occur within the window of the 'refresh-check-delay' value. It is equally important to understand that changes to the script are not actually 'picked up' until a method is called on the dynamic-language-backed bean. It is only when a method is called on a dynamic-language-backed bean that it checks to see if its underlying script source has changed. Any exceptions relating to refreshing the script (such as encountering a compilation error, or finding that the script file has been deleted) will result in a fatal exception being propagated to the calling code.

The refreshable bean behavior described above does not apply to dynamic language source files defined using the <lang:inline-script/> element notation (see Inline dynamic language source files). Additionally, it only applies to beans where changes to the underlying source file can actually be detected; for example, by code that checks the last modified date of a dynamic language source file that exists on the filesystem.

Inline dynamic language source files

The dynamic language support can also cater for dynamic language source files that are embedded directly in Spring bean definitions. More specifically, the <lang:inline-script/> element allows you to define dynamic language source immediately inside a Spring configuration file. An example will perhaps make the inline script feature crystal clear:

|  |
| --- |
| <lang:groovy id="messenger">  <lang:inline-script>  package org.springframework.scripting.groovy;  import org.springframework.scripting.Messenger  class GroovyMessenger implements Messenger {  String message  }  </lang:inline-script>  <lang:property name="message" value="I Can Do The Frug" />  </lang:groovy> |

If we put to one side the issues surrounding whether it is good practice to define dynamic language source inside a Spring configuration file, the <lang:inline-script/> element can be useful in some scenarios. For instance, we might want to quickly add a Spring Validator implementation to a Spring MVC Controller. This is but a moment’s work using inline source. (See Scripted Validators for such an example.)

Understanding Constructor Injection in the context of dynamic-language-backed beans

There is one very important thing to be aware of with regard to Spring’s dynamic language support. Namely, it is not (currently) possible to supply constructor arguments to dynamic-language-backed beans (and hence constructor-injection is not available for dynamic-language-backed beans). In the interests of making this special handling of constructors and properties 100% clear, the following mixture of code and configuration will not work.

|  |
| --- |
| // from the file 'Messenger.groovy'  package org.springframework.scripting.groovy;  import org.springframework.scripting.Messenger  class GroovyMessenger implements Messenger {  GroovyMessenger() {}  // this constructor is not available for Constructor Injection  GroovyMessenger(String message) {  this.message = message;  }  String message  String anotherMessage  } |

|  |
| --- |
| <lang:groovy id="badMessenger"  script-source="classpath:Messenger.groovy">  <!-- this next constructor argument will not be injected into the GroovyMessenger -->  <!-- in fact, this isn't even allowed according to the schema -->  <constructor-arg value="This will not work" />  <!-- only property values are injected into the dynamic-language-backed object -->  <lang:property name="anotherMessage" value="Passed straight through to the dynamic-language-backed object" />  </lang> |

In practice this limitation is not as significant as it first appears since setter injection is the injection style favored by the overwhelming majority of developers anyway (let’s leave the discussion as to whether that is a good thing to another day).

##### 3.3.2. Groovy beans

|  |
| --- |
| *The Groovy library dependencies*  *The Groovy scripting support in Spring requires the following libraries to be on the classpath of your application.*  *groovy-1.8.jar*  *asm-3.2.jar*  *antlr-2.7.7.jar* |

From the Groovy homepage…​

"Groovy is an agile dynamic language for the Java 2 Platform that has many of the features that people like so much in languages like Python, Ruby and Smalltalk, making them available to Java developers using a Java-like syntax. "

If you have read this chapter straight from the top, you will already have seen an example of a Groovy-dynamic-language-backed bean. Let’s look at another example (again using an example from the Spring test suite).

|  |
| --- |
| package org.springframework.scripting;  public interface Calculator {  int add(int x, int y);  } |

Here is an implementation of the Calculator interface in Groovy.

|  |
| --- |
| // from the file 'calculator.groovy'  package org.springframework.scripting.groovy  class GroovyCalculator implements Calculator {  int add(int x, int y) {  x + y  }  } |

|  |
| --- |
| <-- from the file 'beans.xml' -->  <beans>  <lang:groovy id="calculator" script-source="classpath:calculator.groovy"/>  </beans> |

Lastly, here is a small application to exercise the above configuration.

|  |
| --- |
| package org.springframework.scripting;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  public class Main {  public static void Main(String[] args) {  ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");  Calculator calc = (Calculator) ctx.getBean("calculator");  System.out.println(calc.add(2, 8));  }  } |

The resulting output from running the above program will be (unsurprisingly) 10. (Exciting example, huh? Remember that the intent is to illustrate the concept. Please consult the dynamic language showcase project for a more complex example, or indeed Scenarios later in this chapter).

It is important that you do not define more than one class per Groovy source file. While this is perfectly legal in Groovy, it is (arguably) a bad practice: in the interests of a consistent approach, you should (in the opinion of this author) respect the standard Java conventions of one (public) class per source file.

Customizing Groovy objects via a callback

The GroovyObjectCustomizer interface is a callback that allows you to hook additional creation logic into the process of creating a Groovy-backed bean. For example, implementations of this interface could invoke any required initialization method(s), or set some default property values, or specify a custom MetaClass.

|  |
| --- |
| public interface GroovyObjectCustomizer {  void customize(GroovyObject goo);  } |

The Spring Framework will instantiate an instance of your Groovy-backed bean, and will then pass the created GroovyObject to the specified GroovyObjectCustomizer if one has been defined. You can do whatever you like with the supplied GroovyObject reference: it is expected that the setting of a custom MetaClass is what most folks will want to do with this callback, and you can see an example of doing that below.

|  |
| --- |
| public final class SimpleMethodTracingCustomizer implements GroovyObjectCustomizer {  public void customize(GroovyObject goo) {  DelegatingMetaClass metaClass = new DelegatingMetaClass(goo.getMetaClass()) {  public Object invokeMethod(Object object, String methodName, Object[] arguments) {  System.out.println("Invoking '" + methodName + "'.");  return super.invokeMethod(object, methodName, arguments);  }  };  metaClass.initialize();  goo.setMetaClass(metaClass);  }  } |

A full discussion of meta-programming in Groovy is beyond the scope of the Spring reference manual. Consult the relevant section of the Groovy reference manual, or do a search online: there are plenty of articles concerning this topic. Actually making use of a GroovyObjectCustomizer is easy if you are using the Spring namespace support.

|  |
| --- |
| <!-- define the GroovyObjectCustomizer just like any other bean -->  <bean id="tracingCustomizer" class="example.SimpleMethodTracingCustomizer"/>  <!-- ... and plug it into the desired Groovy bean via the 'customizer-ref' attribute -->  <lang:groovy id="calculator"  script-source="classpath:org/springframework/scripting/groovy/Calculator.groovy"  customizer-ref="tracingCustomizer"/> |

If you are not using the Spring namespace support, you can still use the GroovyObjectCustomizer functionality.

|  |
| --- |
| <bean id="calculator" class="org.springframework.scripting.groovy.GroovyScriptFactory">  <constructor-arg value="classpath:org/springframework/scripting/groovy/Calculator.groovy"/>  <!-- define the GroovyObjectCustomizer (as an inner bean) -->  <constructor-arg>  <bean id="tracingCustomizer" class="example.SimpleMethodTracingCustomizer"/>  </constructor-arg>  </bean>  <bean class="org.springframework.scripting.support.ScriptFactoryPostProcessor"/> |

|  |
| --- |
| *As of Spring Framework 4.3.3, you may also specify a Groovy CompilationCustomizer (such as an ImportCustomizer) or even a full Groovy CompilerConfiguration object in the same place as Spring’s GroovyObjectCustomizer.* |

##### 3.3.3. BeanShell beans

|  |
| --- |
| *The BeanShell library dependencies*  *The BeanShell scripting support in Spring requires the following libraries to be on the classpath of your application.*  *bsh-2.0b4.jar* |

From the BeanShell homepage…​

"BeanShell is a small, free, embeddable Java source interpreter with dynamic language features, written in Java. BeanShell dynamically executes standard Java syntax and extends it with common scripting conveniences such as loose types, commands, and method closures like those in Perl and JavaScript."

In contrast to Groovy, BeanShell-backed bean definitions require some (small) additional configuration. The implementation of the BeanShell dynamic language support in Spring is interesting in that what happens is this: Spring creates a JDK dynamic proxy implementing all of the interfaces that are specified in the 'script-interfaces' attribute value of the <lang:bsh> element (this is why you must supply at least one interface in the value of the attribute, and (accordingly) program to interfaces when using BeanShell-backed beans). This means that every method call on a BeanShell-backed object is going through the JDK dynamic proxy invocation mechanism.

Let’s look at a fully working example of using a BeanShell-based bean that implements the Messenger interface that was defined earlier in this chapter (repeated below for your convenience).

|  |
| --- |
| package org.springframework.scripting;  public interface Messenger {  String getMessage();  } |

Here is the BeanShell 'implementation' (the term is used loosely here) of the Messenger interface.

|  |
| --- |
| String message;  String getMessage() {  return message;  }  void setMessage(String aMessage) {  message = aMessage;  } |

And here is the Spring XML that defines an 'instance' of the above 'class' (again, the term is used very loosely here).

|  |
| --- |
| <lang:bsh id="messageService" script-source="classpath:BshMessenger.bsh"  script-interfaces="org.springframework.scripting.Messenger">  <lang:property name="message" value="Hello World!" />  </lang:bsh> |

See Scenarios for some scenarios where you might want to use BeanShell-based beans.

#### 3.4. Scenarios

The possible scenarios where defining Spring managed beans in a scripting language would be beneficial are, of course, many and varied. This section describes two possible use cases for the dynamic language support in Spring.

##### 3.4.1. Scripted Spring MVC Controllers

One group of classes that may benefit from using dynamic-language-backed beans is that of Spring MVC controllers. In pure Spring MVC applications, the navigational flow through a web application is to a large extent determined by code encapsulated within your Spring MVC controllers. As the navigational flow and other presentation layer logic of a web application needs to be updated to respond to support issues or changing business requirements, it may well be easier to effect any such required changes by editing one or more dynamic language source files and seeing those changes being immediately reflected in the state of a running application.

Remember that in the lightweight architectural model espoused by projects such as Spring, you are typically aiming to have a really thin presentation layer, with all the meaty business logic of an application being contained in the domain and service layer classes. Developing Spring MVC controllers as dynamic-language-backed beans allows you to change presentation layer logic by simply editing and saving text files; any changes to such dynamic language source files will (depending on the configuration) automatically be reflected in the beans that are backed by dynamic language source files.

|  |
| --- |
| *In order to effect this automatic 'pickup' of any changes to dynamic-language-backed beans, you will have had to enable the 'refreshable beans' functionality. See Refreshable beans for a full treatment of this feature.* |

Find below an example of an org.springframework.web.servlet.mvc.Controller implemented using the Groovy dynamic language.

|  |
| --- |
| // from the file '/WEB-INF/groovy/FortuneController.groovy'  package org.springframework.showcase.fortune.web  import org.springframework.showcase.fortune.service.FortuneService  import org.springframework.showcase.fortune.domain.Fortune  import org.springframework.web.servlet.ModelAndView  import org.springframework.web.servlet.mvc.Controller  import javax.servlet.http.HttpServletRequest  import javax.servlet.http.HttpServletResponse  class FortuneController implements Controller {  @Property FortuneService fortuneService  ModelAndView handleRequest(HttpServletRequest request,  HttpServletResponse httpServletResponse) {  return new ModelAndView("tell", "fortune", this.fortuneService.tellFortune())  }  } |

|  |
| --- |
| <lang:groovy id="fortune"  refresh-check-delay="3000"  script-source="/WEB-INF/groovy/FortuneController.groovy">  <lang:property name="fortuneService" ref="fortuneService"/>  </lang:groovy> |

##### 3.4.2. Scripted Validators

Another area of application development with Spring that may benefit from the flexibility afforded by dynamic-language-backed beans is that of validation. It may be easier to express complex validation logic using a loosely typed dynamic language (that may also have support for inline regular expressions) as opposed to regular Java.

Again, developing validators as dynamic-language-backed beans allows you to change validation logic by simply editing and saving a simple text file; any such changes will (depending on the configuration) automatically be reflected in the execution of a running application and would not require the restart of an application.

|  |
| --- |
| *Please note that in order to effect the automatic 'pickup' of any changes to dynamic-language-backed beans, you will have had to enable the 'refreshable beans' feature. See Refreshable beans for a full and detailed treatment of this feature.* |

Find below an example of a Spring org.springframework.validation.Validator implemented using the Groovy dynamic language. (See Validation using Spring’s Validator interface for a discussion of the Validator interface.)

|  |
| --- |
| import org.springframework.validation.Validator  import org.springframework.validation.Errors  import org.springframework.beans.TestBean  class TestBeanValidator implements Validator {  boolean supports(Class clazz) {  return TestBean.class.isAssignableFrom(clazz)  }  void validate(Object bean, Errors errors) {  if(bean.name?.trim()?.size() > 0) {  return  }  errors.reject("whitespace", "Cannot be composed wholly of whitespace.")  }  } |

#### 3.5. Bits and bobs

This last section contains some bits and bobs related to the dynamic language support.

##### 3.5.1. AOP - advising scripted beans

It is possible to use the Spring AOP framework to advise scripted beans. The Spring AOP framework actually is unaware that a bean that is being advised might be a scripted bean, so all of the AOP use cases and functionality that you may be using or aim to use will work with scripted beans. There is just one (small) thing that you need to be aware of when advising scripted beans…​ you cannot use class-based proxies, you must use interface-based proxies.

You are of course not just limited to advising scripted beans…​ you can also write aspects themselves in a supported dynamic language and use such beans to advise other Spring beans. This really would be an advanced use of the dynamic language support though.

##### 3.5.2. Scoping

In case it is not immediately obvious, scripted beans can of course be scoped just like any other bean. The scope attribute on the various <lang:language/> elements allows you to control the scope of the underlying scripted bean, just as it does with a regular bean. (The default scope is singleton, just as it is with 'regular' beans.)

Find below an example of using the scope attribute to define a Groovy bean scoped as a prototype.

|  |
| --- |
| <?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:lang="http://www.springframework.org/schema/lang"  xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang.xsd">  <lang:groovy id="messenger" script-source="classpath:Messenger.groovy" scope="prototype">  <lang:property name="message" value="I Can Do The RoboCop" />  </lang:groovy>  <bean id="bookingService" class="x.y.DefaultBookingService">  <property name="messenger" ref="messenger" />  </bean>  </beans> |

See Bean scopes in The IoC container for a fuller discussion of the scoping support in the Spring Framework.

##### 3.5.3. The lang XML schema

The lang tags in Spring XML configuration deal with exposing objects that have been written in a dynamic language such as JRuby or Groovy as beans in the Spring container.

These tags (and the dynamic language support) are comprehensively covered in the chapter entitled Dynamic language support. Please do consult that chapter for full details on this support and the lang tags themselves.

In the interest of completeness, to use the tags in the lang schema, you need to have the following preamble at the top of your Spring XML configuration file; the text in the following snippet references the correct schema so that the tags in the lang namespace are available to you.

|  |
| --- |
| <?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:lang="http://www.springframework.org/schema/lang" xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang.xsd"> <!-- bean definitions here -->  </beans> |

#### 3.6. Further Resources

Find below links to further resources about the various dynamic languages described in this chapter.

The JRuby homepage

The Groovy homepage

The BeanShell homepage