The Spring 4.3 release brought some nice refinements into core container, caching, JMS, Web MVC and testing submodules of the framework.

In this post, we will discuss few of these improvements including:

* Implicit Constructor Injection
* Java 8 Default Interface Methods Support
* Improved Resolution of Dependencies
* Cache Abstraction Refinements
* Composed *@RequestMapping* Variants
* @Requestscope, @Sessionscope, @Applicationscope Annotations
* *@RequestAttribute* and *@SessionAttribute* annotations
* Libraries/Application Servers Versions Support
* the *InjectionPoint* class

**2. Implicit Constructor Injection**

Consider the following service class:

**@Service**

publicclassFooService **{**

privatefinal **FooRepository repository;**

**@Autowired**

publicFooService**(FooRepository repository) {**

this**.repository = repository**

**}**

**}**

Quite a common use case, but if you forget the *@Autowired* annotation on the constructor, the container will throw an exception looking for a default constructor, unless you explicitly do the wiring.

So as of 4.3, you no longer need to specify an explicit injection annotation in such a single-constructor scenario. This is particularly elegant for classes which do not carry any annotations at all:

publicclassFooService **{**

privatefinal **FooRepository repository;**

publicFooService**(FooRepository repository) {**

this**.repository = repository**

**}**

**}**

* In Spring 4.2 and below, the following configuration for this bean will not work, because Spring will not be able to find a default constructor for *FooService*. Spring 4.3 is smarter and will autowire the constructor automatically:

**<beans>**

**<bean** class**="com.baeldung.spring43.ctor.FooRepository" />**

**<bean** class**="com.baeldung.spring43.ctor.FooService" />**

**</beans>**

* Similarly, you may have noticed that *@Configuration* classes historically did not support constructor injection. As of 4.3, they do, and they naturally allow omitting *@Autowired* in a single-constructor scenario as well:

**@Configuration**

publicclassFooConfiguration **{**

privatefinal **FooRepository repository;**

publicFooConfiguration**(FooRepository repository) {**

this**.repository = repository;**

**}**

**@Bean**

public **FooService** fooService**() {**

returnnew **FooService(**this**.repository);**

**}**

**}**

* **3. Java 8 Default Interface Methods Support**

Before Spring 4.3, default interface methods were not supported.

This was not easy to implement because even JDK's JavaBean introspector did not detect default methods as accessors. Since Spring 4.3, getters and setters implemented as default interface methods are identified during injection, which allows to use them for instance as common preprocessors for accessed properties, like in this example:

publicinterfaceIDateHolder **{**

voidsetLocalDate**(LocalDate localDate);**

**LocalDate** getLocalDate**();**

defaultvoidsetStringDate**(String stringDate) {**

**setLocalDate(LocalDate.parse(stringDate,**

**DateTimeFormatter.ofPattern("dd.MM.yyyy")));**

**}**

**}**

**This bean may now have the stringDate property injected:**

**<bean id="dateHolder"**

class**="com.baeldung.spring43.defaultmethods.DateHolder">**

**<property name="stringDate" value="15.10.1982"/>**

**</bean>**

* Same goes for using test annotations like *@BeforeTransaction* and *@AfterTransaction*on default interface methods. JUnit 5 already supports its test annotations on default interface methods, and Spring 4.3 follows the lead. Now you can abstract common testing logic in an interface and implement it in test classes. Here is an interface for test cases that logs messages before and after transactions in tests:

publicinterfaceITransactionalTest **{**

**Logger log = LoggerFactory.getLogger(ITransactionalTest.class);**

**@BeforeTransaction**

defaultvoidbeforeTransaction**() {**

**log.info("Before opening transaction");**

**}**

**@AfterTransaction**

defaultvoidafterTransaction**() {**

**log.info("After closing transaction");**

**}**

**}**

Another improvement concerning annotations *@BeforeTransaction,* *@AfterTransaction* and *@Transactional* is the relaxation of the requirement that the annotated methods should be *public* — now they may have any visibility level.

* **4. Improved Resolution of Dependencies**

The newest version also introduces the *ObjectProvider*, an extension of the existing *ObjectFactory* interface with handy signatures such as *getIfAvailable* and *getIfUnique* to retrieve a bean only if it exists or if a single candidate can be determined (in particular: a primary candidate in case of multiple matching beans).

**@Service**

publicclassFooService **{**

privatefinal **FooRepository repository;**

publicFooService**(ObjectProvider<FooRepository> repositoryProvider) {**

this**.repository = repositoryProvider.getIfUnique();**

**}**

**}**

You may use such *ObjectProvider* handle for custom resolution purposes during initialization as shown above, or store the handle in a field for late on-demand resolution (as you typically do with an *ObjectFactory*).

* **5. Cache Abstraction Refinements**

The cache abstraction is mainly used to cache values that are CPU and IO consuming. In particular use cases, a given key may be requested by several threads (i.e. clients) in parallel, especially on startup. Synchronized cache support is a long-requested feature that has now been implemented. Assume the following:

@Service

**public** **class** **FooService** {

@Cacheable(cacheNames = "foos", sync = true)

**public** Foo **getFoo**(String id) { ... }

}

Notice the sync = true attribute which tells the framework to block any concurrent threads while the value is being computed. This will make sure that this intensive operation is invoked only once in case of concurrent access.

Spring 4.3 also improves the caching abstraction as follows:

* SpEL expressions in cache-related annotations can now refer to beans (i.e. @beanName.method()).
* ConcurrentMapCacheManager and ConcurrentMapCache now support the serialization of cache entries via a new storeByValue attribute.
* @Cacheable, @CacheEvict, @CachePut, and @Caching may now be used as meta-annotations to create custom composed annotations with attribute overrides.

## ****6. Composed****@RequestMapping****Variants****

Spring Framework 4.3 introduces the following method-level composed variants of the @RequestMapping annotation that help to simplify mappings for common HTTP methods and better express the semantics of the annotated handler method.

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping
* @PatchMapping

For example, @GetMapping is a shorter form of saying @RequestMapping(method = RequestMethod.GET). The following example shows an MVC controller that has been simplified with a composed @GetMapping annotation.

@Controller

@RequestMapping("/appointments")

**public** **class** **AppointmentsController** {

**private** **final** AppointmentBook appointmentBook;

@Autowired

**public** **AppointmentsController**(AppointmentBook appointmentBook) {

**this**.appointmentBook = appointmentBook;

}

@GetMapping

**public** Map<String, Appointment> **get**() {

**return** appointmentBook.getAppointmentsForToday();

}

}

## ****7.****@RequestScope****,****@SessionScope****,****@ApplicationScope****Annotations****

When using annotation-driven components or Java Config, the @RequestScope, @SessionScope and @ApplicationScope annotations can be used to assign a component to the required scope. These annotations not only set the scope of the bean but also set the scoped proxy mode to ScopedProxyMode.TARGET\_CLASS.

TARGET\_CLASS mode means that CGLIB proxy will be used for proxying of this bean and ensuring that it can be injected in any other bean, even with a broader scope. TARGET\_CLASS mode allows proxying not only for interfaces but classes too.

@RequestScope

@Component

**public** **class** **LoginAction** {

// ...

}

@SessionScope

@Component

**public** **class** **UserPreferences** {

// ...

}

@ApplicationScope

@Component

**public** **class** **AppPreferences** {

// ...

}

## 8. @RequestAttribute and @SessionAttribute Annotations

Two more annotations for injecting parameters of the HTTP request into Controller methods appeared, namely @RequestAttribute and @SessionAttribute. They allow you to access some pre-existing attributes, managed globally (i.e. outside the Controller). The values for these attributes may be provided, for instance, by registered instances of javax.servlet.Filter or org.springframework.web.servlet.HandlerInterceptor.

Suppose we have registered the following HandlerInterceptor implementation that parses the request and adds login parameter to the session and another query parameter to a request:

**public** **class** **ParamInterceptor** **extends** **HandlerInterceptorAdapter** {

@Override

**public** **boolean** **preHandle**(HttpServletRequest request,

HttpServletResponse response, Object handler) **throws** Exception {

request.getSession().setAttribute("login", "john");

request.setAttribute("query", "invoices");

**return** **super**.preHandle(request, response, handler);

}

}

Such parameters may be injected into a Controller instance with corresponding annotations on method arguments:

@GetMapping

**public** String **get**(@SessionAttribute String login,

@RequestAttribute String query) {

**return** String.format("login = %s, query = %s", login, query);

}

## ****9. Libraries/Application Servers Versions Support****

Spring 4.3 supports the following library versions and server generations:

* Hibernate ORM 5.2 (still supporting 4.2/4.3 and 5.0/5.1 as well, with 3.6 deprecated now)
* Jackson 2.8 (minimum raised to Jackson 2.6+ as of Spring 4.3)
* OkHttp 3.x (still supporting OkHttp 2.x side by side)
* Netty 4.1
* Undertow 1.4
* Tomcat 8.5.2 as well as 9.0 M6

Furthermore, Spring 4.3 embeds the updated ASM 5.1 and Objenesis 2.4 in spring-core.jar.

## ****10.****InjectionPoint

The InjectionPoint class is a new class introduced in Spring 4.3 which **provides information about places where a particular bean gets injected**, whether it is a method/constructor parameter or a field.

The types of information you can find using this class are:

* Field object – you can obtain the point of injection wrapped as a Field object by using the getField() method if the bean is injected into a field
* MethodParameter – you can call getMethodParameter() method to obtain the injection point wrapped as a MethodParameter object if the bean is injected into a parameter
* Member – calling getMember() method will return the entity containing the injected bean wrapped into a Member object
* Class<?> – obtain the declared type of the parameter or field where the bean in injected, using getDeclaredType()
* Annotation[] – by using the getAnnotations() method, you can retrieve an array of Annotation objects which represent the annotations associated with the field or parameter
* AnnotatedElement – call getAnnotatedElement() to get the injection point wrapped as an AnnotatedElement object

A case in which this class is very useful is when we want to create Logger beans based on the class to which they belong:

@Bean

@Scope("prototype")

**public** Logger **logger**(InjectionPoint injectionPoint) {

**return** Logger.getLogger(

injectionPoint.getMethodParameter().getContainingClass());

}

The bean has to be defined with a prototype scope so that a different logger is created for each class. If you create a singleton bean and inject in multiple places, the Spring will return the first encountered injection point.

Then, we can inject the bean into our AppointmentsController:

@Autowired

**private** Logger logger;

## ****11. Conclusion****

In this article, we discussed some of the new features introduced with Spring 4.3.

We've covered useful annotations that eliminate boilerplate, new helpful methods of dependency lookup and injection and several substantial improvements within the web and caching facilities.