Language Support

Version 5.0.4.RELEASE

Table of Contents

Chapter 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.

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.

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()
```

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 primary constructor properties, make sure to use annotation use-site targets 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

`WebTestClient` is not usable yet in Kotlin due to a type inference issue which is expected to be fixed as of Kotlin 1.3. You can watch SPR-16057 for up-to-date information. Meanwhile, the proposed alternative is to use directly `WebClient` with its Reactor and Spring Kotlin extensions to perform integration tests on an embedded WebFlux server.

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. 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

1.10.2. 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

1.10.3. Tutorials

- Creating a RESTful Web Service with Spring Boot

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
- JSR-223 application classpath not available when using Java 9
- 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

Chapter 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.

Chapter 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 <strong>not</strong> be injected into the
GroovyMessenger -->
    <!-- in fact, this isn't even allowed according to the schema -->
    <constructor-arg value="This will <strong>not</strong> 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"
    <em>xmlns:lang="http://www.springframework.org/schema/lang"</em>
xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        <em>http://www.springframework.org/schema/lang
http://www.springframework.org/schema/lang/spring-lang.xsd"</em>> <!-- 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