Language Support

Version 5.2.8.BUILD-SNAPSHOT

Table of Contents

1. Kotlin . . . 1
   1.1. Requirements . . . 1
   1.2. Extensions . . . 1
   1.3. Null-safety . . . 2
   1.4. Classes and Interfaces . . . 2
   1.5. Annotations . . . 3
   1.6. Bean Definition DSL . . . 3
   1.7. Web . . . 5
      1.7.1. Router DSL . . . 6
      1.7.2. MockMvc DSL . . . 6
      1.7.3. Kotlin Script Templates . . . 7
   1.8. Coroutines . . . 8
      1.8.1. Dependencies . . . 8
      1.8.2. How Reactive translates to Coroutines? . . . 9
      1.8.3. Controllers . . . 9
      1.8.4. WebFlux.fn . . . 11
      1.8.5. Transactions . . . 12
   1.9. Spring Projects in Kotlin . . . 13
      1.9.1. Final by Default . . . 13
      1.9.2. Using Immutable Class Instances for Persistence . . . 14
      1.9.3. Injecting Dependencies . . . 15
      1.9.4. Injecting Configuration Properties . . . 15
      1.9.5. Checked Exceptions . . . 16
      1.9.6. Annotation Array Attributes . . . 16
      1.9.7. Testing . . . 17
         Constructor injection . . . 17
         `PER_CLASS` Lifecycle . . . 18
         Specification-like Tests . . . 19
         `WebTestClient` Type Inference Issue in Kotlin . . . 19
   1.10. Getting Started . . . 19
      1.10.1. `start.spring.io` . . . 19
      1.10.2. Choosing the Web Flavor . . . 19
   1.11. Resources . . . 20
      1.11.1. Examples . . . 20
      1.11.2. Issues . . . 20
2. Apache Groovy . . . 22
3. Dynamic Language Support . . . 23
   3.1. A First Example . . . 23

3.2. Defining Beans that Are Backed by Dynamic Languages . . . 25
3.2.1. Common Concepts . . . 25
The <lang:language/> element . . . 26
Refreshable Beans . . . 26
Inline Dynamic Language Source Files . . . 29
Understanding Constructor Injection in the Context of Dynamic-language-backed Beans . . . 29
3.2.2. Groovy Beans . . . 30
Customizing Groovy Objects by Using a Callback . . . 32
3.2.3. BeanShell Beans . . . 33
3.3. Scenarios . . . 34
3.3.1. Scripted Spring MVC Controllers . . . 35
3.3.2. Scripted Validators . . . 36
3.4. Additional Details . . . 37
3.4.1. AOP — Advising Scripted Beans . . . 37
3.4.2. Scoping . . . 37
3.4.3. The `lang` XML schema . . . 38
3.5. Further Resources . . . 38

Chapter 1. Kotlin

Kotlin is a statically typed language that targets 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 and lets developers write Kotlin applications almost as if the Spring Framework was a native Kotlin framework.

The easiest way to build a Spring application with Kotlin is to leverage Spring Boot and its dedicated Kotlin support. This comprehensive tutorial will teach you how to build Spring Boot applications with Kotlin using start.spring.io.

As of Spring Framework 5.2, most of the code samples of the reference documentation are provided in Kotlin in addition to Java.

Feel free to join the #spring channel of Kotlin Slack or ask a question with `spring` and `kotlin` as tags on Stackoverflow if you need support.

1.1. Requirements

Spring Framework supports Kotlin 1.3 and requires `kotlin-stdlib` (or one of its variants, such as `kotlin-stdlib-jdk8`) and `kotlin-reflect` to be present on the classpath. They are provided by default if you bootstrap 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 use these extensions to add new Kotlin-specific conveniences to existing Spring APIs.

The Spring Framework KDoc API lists and documents all available the Kotlin extensions and DSLs.



Keep in mind that Kotlin extensions need to be imported to be used. This means, for example, that the `GenericApplicationContext.registerBean` Kotlin extension is available only if `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 the Spring Framework provides some extensions to take advantage of this feature. This allows for a better Kotlin API `RestTemplate`, for the new `WebClient` from Spring WebFlux, and for various other APIs.



Other libraries, such as Reactor and Spring Data, also provide Kotlin extensions for their APIs, thus giving a better Kotlin development experience overall.

To retrieve a list of `User` objects in Java, you would normally write the following:

```
Flux<User> users  = client.get().retrieve().bodyToFlux(User.class)
```

With Kotlin and the Spring Framework extensions, you can instead write the following:

```
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, such as `Optional`. (Kotlin allows using functional constructs with nullable values. See this comprehensive guide to Kotlin null-safety.)

Although Java does not let you express null-safety in its type-system, the Spring Framework 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 and 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 such as Reactor or Spring Data provide null-safe APIs to leverage this feature.

You can configure JSR-305 checks 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 as `-Xjsr305=warn`. The `strict` value is required to have Spring Framework API null-safety taken into 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 that 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 and Interfaces

The Spring Framework supports various Kotlin constructs, such as instantiating Kotlin classes through primary constructors, immutable classes data binding, and function optional parameters

with default values.

Kotlin parameter names are recognized through a dedicated `KotlinReflectionParameterNameDiscoverer`, which allows finding interface method parameter names without requiring the Java 8 `-parameters` compiler flag to be enabled during compilation.

The Jackson Kotlin module, which is required for serializing or deserializing JSON data, is automatically registered when found in the classpath, and a warning message is logged if Jackson and Kotlin are detected without the Jackson Kotlin module being present.

You can declare configuration classes as top level or nested but not inner, since the later requires a reference to the outer class.

1.5. Annotations

The Spring Framework also takes advantage of Kotlin null-safety to determine if an HTTP parameter is required without having to explicitly define the `required` attribute. That means `@RequestParam name: String?` is treated as not required and, conversely, `@RequestParam name: String` is treated 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 thing: Thing` implies that a bean of type `Thing` must be registered in the application context, while `@Autowired lateinit var thing: Thing?` does not raise an error if such a bean does not exist.

Following the same principle, `@Bean fun play(toy: Toy, car: Car?) = Baz(toy, Car)` implies that a bean of type `Toy` must be registered in the application context, while a bean of type `Car` may or may not exist. The same behavior applies to autowired constructor parameters.



If you use bean validation on classes with properties or a primary constructor parameters, you may need to use annotation use-site targets, such as `@field:NotNull` or `@get:Size(min=5, max=15)`, as described in this Stack Overflow response.

1.6. Bean Definition DSL

Spring Framework supports registering beans in a functional way by using lambdas as an alternative to XML or Java configuration (`@Configuration` and `@Bean`). In a nutshell, it lets you 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, you can, for example, write the following:

```
class Foo {}

class Bar {
    private final Foo foo;
    public Bar(Foo foo) {
        this.foo = foo;
    }
}

GenericApplicationContext context = new GenericApplicationContext();
context.registerBean(Foo.class);
context.registerBean(Bar.class, () -> new Bar(context.getBean(Foo.class)));
```

In Kotlin, with reified type parameters and `GenericApplicationContext` Kotlin extensions, you can instead write the following:

```
class Foo

class Bar(private val foo: Foo)

val context = GenericApplicationContext().apply {
    registerBean<Foo>()
    registerBean { Bar(it.getBean()) }
}
```

When the class `Bar` has a single constructor, you can even just specify the bean class, the constructor parameters will be autowired by type:

```
val context = GenericApplicationContext().apply {
    registerBean<Foo>()
    registerBean<Bar>()
}
```

In order to allow a more declarative approach and cleaner syntax, Spring Framework provides a Kotlin bean definition DSL It declares an `ApplicationContextInitializer` through a clean declarative API, which lets you deal with profiles and `Environment` for customizing how beans are registered.

In the following example notice that:

- Type inference usually allows to avoid specifying the type for bean references like `ref("bazBean")`
- It is possible to use Kotlin top level functions to declare beans using callable references like `bean(::myRouter)` in this example
- When specifying `bean<Bar>()` or `bean(::myRouter)`, parameters are autowired by type

- The `FooBar` bean will be registered only if the `foobar` profile is active

```
class Foo
class Bar(private val foo: Foo)
class Baz(var message: String = "")
class FooBar(private val baz: Baz)

val myBeans = beans {
    bean<Foo>()
    bean<Bar>()
    bean("bazBean") {
        Baz().apply {
            message = "Hello world"
        }
    }
    profile("foobar") {
        bean { FooBar(ref("bazBean")) }
    }
    bean(::myRouter)
}

fun myRouter(foo: Foo, bar: Bar, baz: Baz) = router {
    // ...
}
```



This DSL is programmatic, meaning it allows custom registration logic of beans through an `if` expression, a `for` loop, or any other Kotlin constructs.

You can then use this `beans()` function to register beans on the application context, as the following example shows:

```
val context = GenericApplicationContext().apply {
    myBeans.initialize(this)
    refresh()
}
```



Spring Boot is based on JavaConfig and does not yet provide specific support for functional bean definition, but you can experimentally use functional bean definitions through Spring Boot's `ApplicationContextInitializer` support. See this Stack Overflow answer for more details and up-to-date information. See also the experimental Kofu DSL developed in Spring Fu incubator.

1.7. Web

1.7.1. Router DSL

Spring Framework comes with a Kotlin router DSL available in 3 flavors:

- WebMvc.fn DSL with router { }
- WebFlux.fn Reactive DSL with router { }
- WebFlux.fn Coroutines DSL with coRouter { }

These DSL let you write clean and idiomatic Kotlin code to build a `RouterFunction` instance as the following example shows:

```
@Configuration
class RouterRouterConfiguration {

    @Bean
    fun mainRouter(userHandler: UserHandler) = 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, meaning that it allows custom registration logic of beans through an `if` expression, a `for` loop, or any other Kotlin constructs. That can be useful when you need to register routes depending on dynamic data (for example, from a database).

See MiXiT project for a concrete example.

1.7.2. MockMvc DSL

A Kotlin DSL is provided via `MockMvc` Kotlin extensions in order to provide a more idiomatic Kotlin API and to allow better discoverability (no usage of static methods).

```
val mockMvc: MockMvc = ...
mockMvc.get("/person/{name}", "Lee") {
    secure = true
    accept = APPLICATION_JSON
    headers {
        contentLanguage = Locale.FRANCE
    }
    principal = Principal { "foo" }
}.andExpect {
    status { isOk }
    content { contentType(APPLICATION_JSON) }
    jsonPath("$.name") { value("Lee") }
    content { json("""{"someBoolean": false}""", false) }
}.andDo {
    print()
}
```

1.7.3. Kotlin Script Templates

Spring Framework provides a `ScriptTemplateView` which supports JSR-223 to render templates by using script engines.

By leveraging `kotlin-script-runtime` and `scripting-jsr223-embeddable` dependencies, it is possible to use such feature to render Kotlin-based templates with kotlinx.html DSL or Kotlin multiline interpolated `String`.

`build.gradle.kts`

```
dependencies {
    compile("org.jetbrains.kotlin:kotlin-script-runtime:${kotlinVersion}")
    runtime("org.jetbrains.kotlin:kotlin-scripting-jsr223-
embeddable:${kotlinVersion}")
}
```

Configuration is usually done with `ScriptTemplateConfigurer` and `ScriptTemplateViewResolver` beans.

`KotlinScriptConfiguration.kt`

```
@Configuration
class KotlinScriptConfiguration {

    @Bean
    fun kotlinScriptConfigurer() = ScriptTemplateConfigurer().apply {
        engineName = "kotlin"
        setScripts("scripts/render.kts")
        renderFunction = "render"
        isSharedEngine = false
    }

    @Bean
    fun kotlinScriptViewResolver() = ScriptTemplateViewResolver().apply {
        setPrefix("templates/")
        setSuffix(".kts")
    }
}
```

See the kotlin-script-templating example project for more details.

1.8. Coroutines

Kotlin Coroutines are Kotlin lightweight threads allowing to write non-blocking code in an imperative way. On language side, suspending functions provides an abstraction for asynchronous operations while on library side kotlinx.coroutines provides functions like `async { }` and types like `Flow`.

Spring Framework provides support for Coroutines on the following scope:

- Deferred and Flow return values support in Spring WebFlux annotated `@Controller`
- Suspending function support in Spring WebFlux annotated `@Controller`
- Extensions for WebFlux client and server functional API.
- WebFlux.fn coRouter { } DSL
- Suspending function and `Flow` support in RSocket `@MessageMapping` annotated methods
- Extensions for `RSocketRequester`

1.8.1. Dependencies

Coroutines support is enabled when `kotlinx-coroutines-core` and `kotlinx-coroutines-reactor` dependencies are in the classpath:

`build.gradle.kts`

```
dependencies {

    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
core:${coroutinesVersion}")
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-
reactor:${coroutinesVersion}")
}
```

Version `1.3.0` and above are supported.

1.8.2. How Reactive translates to Coroutines?

For return values, the translation from Reactive to Coroutines APIs is the following:

- `fun handler(): Mono<Void>` becomes `suspend fun handler()`
- `fun handler(): Mono<T>` becomes `suspend fun handler(): T` or `suspend fun handler(): T?` depending on if the `Mono` can be empty or not (with the advantage of being more statically typed)
- `fun handler(): Flux<T>` becomes `fun handler(): Flow<T>`

For input parameters:

- If laziness is not needed, `fun handler(mono: Mono<T>)` becomes `fun handler(value: T)` since a suspending functions can be invoked to get the value parameter.
- If laziness is needed, `fun handler(mono: Mono<T>)` becomes `fun handler(supplier: suspend () → T)` or `fun handler(supplier: suspend () → T?)`

`Flow` is `Flux` equivalent in Coroutines world, suitable for hot or cold stream, finite or infinite streams, with the following main differences:

- `Flow` is push-based while `Flux` is push-pull hybrid
- Backpressure is implemented via suspending functions
- `Flow` has only a single suspending `collect` method and operators are implemented as extensions
- Operators are easy to implement thanks to Coroutines
- Extensions allow to add custom operators to `Flow`
- Collect operations are suspending functions
- `map` operator supports asynchronous operation (no need for `flatMap`) since it takes a suspending function parameter

Read this blog post about Going Reactive with Spring, Coroutines and Kotlin Flow for more details, including how to run code concurrently with Coroutines.

1.8.3. Controllers

Here is an example of a Coroutines `@RestController`.

```
```

```
@RestController
class CoroutinesRestController(client: WebClient, banner: Banner) {

    @GetMapping("/suspend")
    suspend fun suspendingEndpoint(): Banner {
        delay(10)
        return banner
    }

    @GetMapping("/flow")
    fun flowEndpoint() = flow {
        delay(10)
        emit(banner)
        delay(10)
        emit(banner)
    }

    @GetMapping("/deferred")
    fun deferredEndpoint() = GlobalScope.async {
        delay(10)
        banner
    }

    @GetMapping("/sequential")
    suspend fun sequential(): List<Banner> {
        val banner1 = client
                .get()
                .uri("/suspend")
                .accept(MediaType.APPLICATION_JSON)
                .awaitExchange()
                .awaitBody<Banner>()
        val banner2 = client
                .get()
                .uri("/suspend")
                .accept(MediaType.APPLICATION_JSON)
                .awaitExchange()
                .awaitBody<Banner>()
        return listOf(banner1, banner2)
    }

    @GetMapping("/parallel")
    suspend fun parallel(): List<Banner> = coroutineScope {
        val deferredBanner1: Deferred<Banner> = async {
            client
                    .get()
                    .uri("/suspend")
                    .accept(MediaType.APPLICATION_JSON)
                    .awaitExchange()
                    .awaitBody<Banner>()
        }
        val deferredBanner2: Deferred<Banner> = async {
```

```
            client
                    .get()
                    .uri("/suspend")
                    .accept(MediaType.APPLICATION_JSON)
                    .awaitExchange()
                    .awaitBody<Banner>()
        }
        listOf(deferredBanner1.await(), deferredBanner2.await())
    }

    @GetMapping("/error")
    suspend fun error() {
        throw IllegalStateException()
    }

    @GetMapping("/cancel")
    suspend fun cancel() {
        throw CancellationException()
    }

}
```

View rendering with a `@Controller` is also supported.

```
@Controller
class CoroutinesViewController(banner: Banner) {

    @GetMapping("/")
    suspend fun render(model: Model): String {
        delay(10)
        model["banner"] = banner
        return "index"
    }
}
```

1.8.4. WebFlux.fn

Here is an example of Coroutines router defined via the coRouter { } DSL and related handlers.

```
@Configuration
class RouterConfiguration {

    @Bean
    fun mainRouter(userHandler: UserHandler) = coRouter {
        GET("/", userHandler::listView)
        GET("/api/user", userHandler::listApi)
    }
}
```

```
class UserHandler(builder: WebClient.Builder) {

    private val client = builder.baseUrl("...").build()

    suspend fun listView(request: ServerRequest): ServerResponse =
            ServerResponse.ok().renderAndAwait("users", mapOf("users" to
            client.get().uri("...").awaitExchange().awaitBody<User>()))

    suspend fun listApi(request: ServerRequest): ServerResponse =

ServerResponse.ok().contentType(MediaType.APPLICATION_JSON).bodyAndAwait(
                client.get().uri("...").awaitExchange().awaitBody<User>())
}
```

1.8.5. Transactions

Transactions on Coroutines are supported via the programmatic variant of the Reactive transaction management provided as of Spring Framework 5.2.

For suspending functions, a `TransactionalOperator.executeAndAwait` extension is provided.

```
import org.springframework.transaction.reactive.executeAndAwait

class PersonRepository(private val operator: TransactionalOperator) {

    suspend fun initDatabase() = operator.executeAndAwait {
        insertPerson1()
        insertPerson2()
    }

    private suspend fun insertPerson1() {
        // INSERT SQL statement
    }

    private suspend fun insertPerson2() {
        // INSERT SQL statement
    }
}
```

For Kotlin `Flow`, a `Flow<T>.transactional` extension is provided.

```
import org.springframework.transaction.reactive.transactional

class PersonRepository(private val operator: TransactionalOperator) {

    fun updatePeople() = findPeople().map(::updatePerson).transactional(operator)

    private fun findPeople(): Flow<Person> {
        // SELECT SQL statement
    }

    private suspend fun updatePerson(person: Person): Person {
        // UPDATE SQL statement
    }
}
```

1.9. Spring Projects in Kotlin

This section provides some specific hints and recommendations worth for developing Spring projects in Kotlin.

1.9.1. 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.

While 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 into consideration. This is because Spring beans (such as `@Configuration` annotated classes which by default need to be extended at runtime for technical reasons) are normally proxied by CGLIB. The workaround is to add an `open` keyword on each class and member function of Spring beans that are proxied by CGLIB, which can quickly become painful and is against the Kotlin principle of keeping code concise and predictable.



It is also possible to avoid CGLIB proxies for configuration classes by using `@Configuration(proxyBeanMethods = false)`. See `proxyBeanMethods` Javadoc for more details.

Fortunately, Kotlin provides a `kotlin-spring` plugin (a preconfigured version of the `kotlin-allopen` plugin) that automatically opens classes and their member functions for types that are annotated or meta-annotated with one of the following annotations:

- `@Component`
- `@Async`
- `@Transactional`
- `@Cacheable`

Meta-annotation 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 the `kotlin-spring` plugin by default. So, in practice, you can write your Kotlin beans without any additional `open` keyword, as in Java.



The Kotlin code samples in Spring Framework documentation do not explicitly specify `open` on the classes and their member functions. The samples are written for projects using the `kotlin-allopen` plugin, since this is the most commonly used setup.

1.9.2. Using Immutable Class Instances for Persistence

In Kotlin, it is convenient and considered to be a 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()` and `hashCode()`
- `toString()` of the form `"User(name=John, age=42)"`
- `componentN()` functions that correspond to the properties in their order of declaration
- `copy()` function

As the following example shows, 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 a workaround for this "default constructor hell", since Kotlin provides a `kotlin-jpa` plugin that 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 uses Spring Data object mappings (such as MongoDB, Redis, Cassandra, and others).

1.9.3. Injecting Dependencies

Our recommendation is to try to favor constructor injection with `val` read-only (and non-nullable when possible) properties, as the following example shows:

```
@Component
class YourBean(
    private val mongoTemplate: MongoTemplate,
    private val solrClient: SolrClient
)
```



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 you really need to use field injection, you can use the `lateinit var` construct, as the following example shows:

```
@Component
class YourBean {

    @Autowired
    lateinit var mongoTemplate: MongoTemplate

    @Autowired
    lateinit var solrClient: SolrClient
}
```

1.9.4. Injecting Configuration Properties

In Java, you can inject configuration properties by using annotations (such as `@Value("${property}")`). However, in Kotlin, `$` is a reserved character that is used for string interpolation.

Therefore, if you wish to use the `@Value` annotation in Kotlin, you need to escape the `$` character by writing `@Value("\${property}")`.



If you use Spring Boot, you should probably use `@ConfigurationProperties` instead of `@Value` annotations.

As an alternative, you can customize the property placeholder prefix by declaring the following configuration beans:

```
@Bean
fun propertyConfigurer() = PropertySourcesPlaceholderConfigurer().apply {
    setPlaceholderPrefix("%{")
}
```

You can customize existing code (such as Spring Boot actuators or `@LocalServerPort`) that uses the `${…}` syntax, with configuration beans, as the following example shows:

```
@Bean
fun kotlinPropertyConfigurer() = PropertySourcesPlaceholderConfigurer().apply {
    setPlaceholderPrefix("%{")
    setIgnoreUnresolvablePlaceholders(true)
}

@Bean
fun defaultPropertyConfigurer() = PropertySourcesPlaceholderConfigurer()
```

1.9.5. Checked Exceptions

Java and Kotlin exception handling are pretty close, with the main difference being that Kotlin treats all exceptions as unchecked exceptions. However, when using proxied objects (for example classes or methods annotated with `@Transactional`), checked exceptions thrown will be wrapped by default in an `UndeclaredThrowableException`.

To get the original exception thrown like in Java, methods should be annotated with `@Throws` to specify explicitly the checked exceptions thrown (for example `@Throws(IOException::class)`).

1.9.6. Annotation Array Attributes

Kotlin annotations are mostly similar to Java annotations, but array attributes (which are extensively used in Spring) behave differently. As explained in the Kotlin documentation you can omit the `value` attribute name, unlike other attributes, and specify it as a `vararg` parameter.

To understand what that means, consider `@RequestMapping` (which is one of the most widely used Spring annotations) as an example. This Java annotation is declared as follows:

```
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, you can specify a single value for the annotation array attribute, and it is automatically converted to an array.

That is why one can write `@RequestMapping(value = "/toys", method = RequestMethod.GET)` or `@RequestMapping(path = "/toys", method = RequestMethod.GET)`.

However, in Kotlin, you must write `@RequestMapping("/toys", method = [RequestMethod.GET])` or `@RequestMapping(path = ["/toys"], 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`, `@PostMapping`, and others.



If the `@RequestMapping method` attribute is not specified, all HTTP methods will be matched, not only the `GET` method.

1.9.7. Testing

This section addresses testing with the combination of Kotlin and Spring Framework. The recommended testing framework is JUnit 5 along with Mockk for mocking.



If you are using Spring Boot, see this related documentation.

Constructor injection

As described in the dedicated section, JUnit 5 allows constructor injection of beans which is pretty useful with Kotlin in order to use `val` instead of `lateinit var`. You can use `@TestConstructor(autowireMode = AutowireMode.ALL)` to enable autowiring for all parameters.

```
@SpringJUnitConfig(TestConfig::class)
@TestConstructor(autowireMode = AutowireMode.ALL)
class OrderServiceIntegrationTests(val orderService: OrderService,
                                   val customerService: CustomerService) {

    // tests that use the injected OrderService and CustomerService
}
```

`PER_CLASS` Lifecycle

Kotlin lets you specify meaningful test function names between backticks (`). As of JUnit 5, Kotlin test classes can use the `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` annotation to enable 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.

You can also change the default behavior to `PER_CLASS` thanks to a `junit-platform.properties` file with a `junit.jupiter.testinstance.lifecycle.default = per_class` property.

The following example demonstrates `@BeforeAll` and `@AfterAll` annotations on non-static methods:

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
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

You can create specification-like tests with JUnit 5 and Kotlin. The following example shows how to do so:

```
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, you must use the Kotlin `expectBody` extension (such as `.expectBody<String>().isEqualTo("toys")`), since it provides a workaround for the Kotlin issue with the Java API.

See also the related SPR-16057 issue.

1.10. Getting Started

The easiest way to learn how to build a Spring application with Kotlin is to follow the dedicated tutorial.

1.10.1. `start.spring.io`

The easiest way to start a new Spring Framework project in Kotlin is to create a new Spring Boot 2 project on start.spring.io.

1.10.2. Choosing the Web Flavor

Spring Framework now comes with two different web stacks: Spring MVC and Spring WebFlux.

Spring WebFlux is recommended if you want to create applications that will deal with latency, long-lived connections, streaming scenarios or if you want to use the web functional Kotlin DSL.

For other use cases, especially if you are using blocking technologies such as JPA, Spring MVC and its annotation-based programming model is the recommended choice.

1.11. Resources

We recommend the following resources for people learning how to build applications with Kotlin and the Spring Framework:

- 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

1.11.1. Examples

The following Github projects offer examples that you can learn from and possibly even extend:

- spring-boot-kotlin-demo: Regular Spring Boot and Spring Data JPA project
- mixit: Spring Boot 2, WebFlux, and Reactive Spring Data MongoDB
- spring-kotlin-functional: Standalone WebFlux and 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 guide for Boot 1.0 and Java to Boot 2.0 and Kotlin
- spring-cloud-gcp-kotlin-app-sample: Spring Boot with Google Cloud Platform Integrations

1.11.2. Issues

The following list categorizes the pending issues related to Spring and Kotlin support:

- Spring Framework
  - Unable to use WebTestClient with mock server in Kotlin
  - Support null-safety at generics, varargs and array elements level
- Kotlin
  - Parent issue for Spring Framework support
  - Kotlin requires type inference where Java doesn't

- Smart cast regression with open classes
- Impossible to pass not all SAM argument as function
- Support JSR 223 bindings directly via script variables
- Kotlin properties do not override Java-style getters and setters

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 Dynamic Language Support.

Chapter 3. Dynamic Language Support

Spring provides comprehensive support for using classes and objects that have been defined by using a dynamic language (such as Groovy) with Spring. This support lets you write any number of classes in a supported dynamic language and have the Spring container transparently instantiate, configure, and dependency inject the resulting objects.

Spring's scripting support primarily targets Groovy and BeanShell. Beyond those specifically supported languages, the JSR-223 scripting mechanism is supported for integration with any JSR-223 capable language provider (as of Spring 4.2), e.g. JRuby.

You can find fully working examples of where this dynamic language support can be immediately useful in Scenarios.

3.1. A First Example

The 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, we 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. If you want to see equivalent examples in any of the other supported languages, take a look at the source code).

The next example shows the `Messenger` interface, which the Groovy bean is going to implement. Note that this interface is defined in plain Java. Dependent objects that are injected with a reference to the `Messenger` do not know that the underlying implementation is a Groovy script. The following listing shows the `Messenger` interface:

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();
}
```

The following example defines 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...
    }
}
```

The following example implements 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
}
```



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 use 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.

Finally, the following example shows the bean definitions that effect the injection of the Groovy-defined `Messenger` implementation into an instance of the `DefaultBookingService` 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:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
https://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 preceding XML snippet is self-explanatory, but do not worry unduly if it is not. Keep reading for the in-depth detail on the whys and wherefores of the preceding configuration.

3.2. 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.

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, we assume that you already know Groovy. If you need further details about the dynamic languages themselves, see Further Resources at the end of this chapter.

3.2.1. Common Concepts

The steps involved in using dynamic-language-backed beans are as follows:

1. Write the test for the dynamic language source code (naturally).
2. Then write the dynamic language source code itself.

3. Define your dynamic-language-backed beans by using the appropriate `<lang:language/>` element in the XML configuration (you can define such beans programmatically by using the Spring API, although you will have to consult the source code for directions on how to do this, as this chapter does not cover this type of advanced configuration). Note that this is an iterative step. You need at least one bean definition for each dynamic language source file (although multiple bean definitions can reference the same source file).

The first two steps (testing and writing your dynamic language source files) are beyond the scope of this chapter. See the language specification and reference manual for your chosen dynamic language and crack on with developing your dynamic language source files. You 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 in the list in the preceding section 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 can 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, e.g. with JRuby)

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 later in this chapter detail this).

Refreshable Beans

One of the (and perhaps the single) 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. 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 you edit and save changes to the file on the file system).

This lets you 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 (later, as requirements change or some other external factor comes into play) edit a dynamic language source file and have any change they make be 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 picks up the new state and logic from the changed dynamic language source file.



This feature is off by default.

Now we can take a look at an example to see how easy it is to start using refreshable beans. To turn on the refreshable beans feature, you 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, the following example shows 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 is 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 do not want the refresh behavior, do not define the attribute.

If we then run the following application, we can exercise the refreshable feature. (Please 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 you (the developer in this scenario) go off and edit the underlying dynamic language source file so that the refresh triggers on the dynamic-language-backed bean when the program resumes execution.

The following listing shows this sample application:

```
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());
    }
}
```

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 quotation marks. The following listing shows the changes that you (the developer) should 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 runs, 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 is `'I Can Do The Frug'` (notice the inclusion of the additional quotation marks).

Changes to a script do not trigger a refresh if the changes occur within the window of the `refresh-check-delay` value. 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 that relate to refreshing the script (such as encountering a compilation error or finding that the script file has been deleted) results in a fatal exception being propagated to the calling code.

The refreshable bean behavior described earlier does not apply to dynamic language source files defined with the `<lang:inline-script/>` element notation (see Inline Dynamic Language Source Files). Additionally, it applies only 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 file system).

Inline Dynamic Language Source Files

The dynamic language support can also cater to dynamic language source files that are embedded directly in Spring bean definitions. More specifically, the `<lang:inline-script/>` element lets you define dynamic language source immediately inside a Spring configuration file. An example might clarify how the inline script feature works:

```
<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, you can not (currently) 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 does not work:

An approach that cannot 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 (we leave the discussion as to whether that is a good thing to another day).

3.2.2. Groovy Beans

This section describes how to use beans defined in Groovy in Spring.

The Groovy homepage includes the following description:

“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 have already seen an example of a Groovy-

dynamic-language-backed bean. Now consider another example (again using an example from the Spring test suite):

```
package org.springframework.scripting;

public interface Calculator {

    int add(int x, int y);
}
```

The following example implements 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
    }
}
```

The following bean definition uses the calculator defined in Groovy:

```
<-- from the file 'beans.xml' -->
<beans>
    <lang:groovy id="calculator" script-source="classpath:calculator.groovy"/>
</beans>
```

Finally, the following small application exercises the preceding 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 is (unsurprisingly) `10`. (For more interesting

examples, see the dynamic language showcase project for a more complex example or see the examples Scenarios later in this chapter).

You must 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 the Spring team) respect the standard Java conventions of one (public) class per source file.

Customizing Groovy Objects by Using a Callback

The `GroovyObjectCustomizer` interface is a callback that lets you hook additional creation logic into the process of creating a Groovy-backed bean. For example, implementations of this interface could invoke any required initialization methods, set some default property values, or specify a custom `MetaClass`. The following listing shows the `GroovyObjectCustomizer` interface definition:

```
public interface GroovyObjectCustomizer {

    void customize(GroovyObject goo);
}
```

The Spring Framework instantiates an instance of your Groovy-backed bean and then passes the created `GroovyObject` to the specified `GroovyObjectCustomizer` (if one has been defined). You can do whatever you like with the supplied `GroovyObject` reference. We expect that most people want to set a custom `MetaClass` with this callback, and the following example shows how to do so:

```
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. See the relevant section of the Groovy reference manual or do a search online. Plenty of articles address this topic. Actually, making use of a `GroovyObjectCustomizer` is easy if you use the Spring namespace support, as the following example shows:

```
<!-- 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 do not use the Spring namespace support, you can still use the `GroovyObjectCustomizer` functionality, as the following example shows:

```
<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"/>
```



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`. Furthermore, you may set a common `GroovyClassLoader` with custom configuration for your beans at the `ConfigurableApplicationContext.setClassLoader` level; this also leads to shared `GroovyClassLoader` usage and is therefore recommendable in case of a large number of scripted beans (avoiding an isolated `GroovyClassLoader` instance per bean).

3.2.3. BeanShell Beans

This section describes how to use BeanShell beans in Spring.

The BeanShell homepage includes the following description:

```
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, because Spring creates a JDK dynamic proxy that implements 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, consequently, program to interfaces when you use BeanShell-backed beans). This means that every method call on a BeanShell-backed object goes through the JDK dynamic proxy invocation mechanism.

Now we can show a fully working example of using a BeanShell-based bean that implements the `Messenger` interface that was defined earlier in this chapter. We again show the definition of the `Messenger` interface:

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();
}
```

The following example shows the BeanShell "implementation" (we use the term loosely here) of the `Messenger` interface:

```
String message;

String getMessage() {
    return message;
}

void setMessage(String aMessage) {
    message = aMessage;
}
```

The following example shows the Spring XML that defines an "instance" of the above "class" (again, we use these terms 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.

3.3. Scenarios

The possible scenarios where defining Spring managed beans in a scripting language would be

beneficial are many and varied. This section describes two possible use cases for the dynamic language support in Spring.

3.3.1. Scripted Spring MVC Controllers

One group of classes that can 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 typically aim 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 lets you change presentation layer logic by editing and saving text files. Any changes to such dynamic language source files is (depending on the configuration) automatically reflected in the beans that are backed by dynamic language source files.



To effect this automatic “pickup” of any changes to dynamic-language-backed beans, you have to enable the “refreshable beans” functionality. See Refreshable Beans for a full treatment of this feature.

The following example shows an `org.springframework.web.servlet.mvc.Controller` implemented by 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>
```

3.3.2. 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 can be easier to express complex validation logic by 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 lets you change validation logic by editing and saving a simple text file. Any such changes is (depending on the configuration) automatically reflected in the execution of a running application and would not require the restart of an application.



To effect the automatic “pickup” of any changes to dynamic-language-backed beans, you have to enable the 'refreshable beans' feature. See Refreshable Beans for a full and detailed treatment of this feature.

The following example shows a Spring `org.springframework.validation.Validator` implemented by 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.")
    }
}
```

3.4. Additional Details

This last section contains some additional details related to the dynamic language support.

3.4.1. AOP — Advising Scripted Beans

You can 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 use (or aim to use) work with scripted beans. When you advise scripted beans, you cannot use class-based proxies. You must use interface-based proxies.

You are not 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.

3.4.2. Scoping

In case it is not immediately obvious, scripted beans can be scoped in the same way as any other bean. The `scope` attribute on the various `<lang:language/>` elements lets you control the scope of the underlying scripted bean, as it does with a regular bean. (The default scope is singleton, as it is with "regular" beans.)

The following example uses 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
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
https://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 full discussion of the scoping support in the Spring Framework.

3.4.3. The `lang` XML schema

The `lang` elements in Spring XML configuration deal with exposing objects that have been written in a dynamic language (such as Groovy or BeanShell) as beans in the Spring container.

These elements (and the dynamic language support) are comprehensively covered in Dynamic Language Support. See that chapter for full details on this support and the `lang` elements.

To use the elements 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
https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
https://www.springframework.org/schema/lang/spring-lang.xsd">

    <!-- bean definitions here -->

</beans>
```

3.5. Further Resources

The following links go to further resources about the various dynamic languages referenced in this chapter:

- The Groovy homepage
- The BeanShell homepage
- The JRuby homepage