# Kotlin support

Version 5.0.0.RELEASE

# Table of Contents

# Chapter 1. Introduction

Kotlin is a statically-typed language targeting the JVM (and other platforms) which allows writing concise and elegant code while providing a very good interoperability with existing libraries written in Java.

Spring Framework 5 introduces first-class support for Kotlin and allows developers to write Spring + Kotlin applications almost as if the Spring Framework was a native Kotlin framework.

# Chapter 2. Requirements

Spring Framework supports Kotlin 1.1+ and requires `kotlin-stdlib` (or one of its `kotlin-stdlib-jre7` / `kotlin-stdlib-jre8` variants) and `kotlin-reflect` to be present on the classpath. They are provided by default if one bootstraps a Kotlin project on start.spring.io.

# Chapter 3. 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 a shorter syntax.

# Chapter 4. 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 1.1.50+, the default behavior is the same to `-Xjsr305=warn`. The `strict` value should be considered experimental (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 SPR-15942 for up-to-date information.

# Chapter 5. 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.

As of Spring Boot 2.0, Jackson Kotlin module is automatically provided via the JSON starter.

# Chapter 6. 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` or `@Inject` uses this information to determine if a bean is required or not. `@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.

# Chapter 7. 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().invoke(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.

# Chapter 8. Web

## 8.1. 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.

## 8.2. 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.

# Chapter 9. Spring projects in Kotlin

This section provides a focus on some specific hints and recommendations worth knowing when developing Spring projects in Kotlin.

## 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.

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 proxified with 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 proxified with CGLIB such as `@Configuration` classes, which can quickly become painful and is against Kotlin principle to keep 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.

## 9.2. Using immutable class instances for persistence

In Kotlin, it is very convenient and 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)
```

But some persistence technologies like 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 kotlin-noarg plugin.

As of Kay release train, Spring Data supports Kotlin immutable class instances and should not require `kotlin-noarg` plugin if the module leverages Spring Data object mapping (like with MongoDB, Redis, Cassandra, etc.).

## 9.3. 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 its parameters automatically autowired, that's why there is no need for `@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
}
```

## 9.4. 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 this:

```
@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 nullable `var` properties (which is far from ideal) 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.

## 9.5. 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 when it is an array attribute it is 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, one will have to write `@RequestMapping("/foo", method = arrayOf(RequestMethod.GET))`. The variant using `path` is not recommended as it need to be written `@RequestMapping(path = arrayOf("/foo"), method = arrayOf(RequestMethod.GET))`.

A workaround for this specific `method` attribute (the most common one) is to use a shortcut annotation such as `@GetMapping` or `@PostMapping`, etc.

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

Improving the syntax and consistency of Kotlin annotation array attributes is discussed in this Kotlin language design issue.

## 9.6. Testing

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()
  }
}
```

# Chapter 10. Getting started

## 10.1. 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.

## 10.2. 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.

# Chapter 11. Resources

- Kotlin language reference
- Kotlin Slack (with a dedicated #spring channel)
- Try Kotlin in your browser
- Kotlin blog
- Awesome Kotlin

## 11.1. 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

## 11.2. 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-petclinic-kotlin: Kotlin version of the Spring PetClinic Sample Application

## 11.3. Tutorials

- Creating a RESTful Web Service with Spring Boot

## 11.4. Issues

Here is a list of pending issues related to Spring + Kotlin support.

### 11.4.1. Spring Framework

- Add support for Kotlin coroutines

### 11.4.2. Spring Boot

- Improve Kotlin support
- Allow `@ConfigurationProperties` binding for immutable POJOs
- Allow `@ConfigurationProperties` binding on interfaces
- Provide support for Kotlin KClass parameter in `SpringApplication.run()`
- Expose the functional bean registration API via `SpringApplication`

### 11.4.3. Kotlin

- Parent issue for Spring Framework support
- Support "::foo" as a short-hand syntax for bound callable reference to "this::foo"
- Allow specifying array annotation attribute single value without arrayOf()
- Kotlin requires type inference where Java doesn’t
- 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