Capítulo 7
Soporte para Kotlin

1. Uso de Kotlin

Spring funciona a la perfección con Kotlin, ofreciendo una integración fluida entre ambos lenguajes. Kotlin, como lenguaje moderno y conciso, proporciona características adicionales en comparación con Java, tales como el tipado *nullable* por defecto, las propiedades, las funciones de extensión y declaraciones de clases de datos, que simplifican y mejoran la legibilidad del código. Gracias a la plena compatibilidad con las bibliotecas Java existentes, Spring puede sacar el máximo partido de Kotlin sin ningún tipo de fricción. Por tanto, los desarrolladores pueden beneficiarse de la potencia y flexibilidad de Kotlin para desarrollar aplicaciones más eficientes y expresivas, al tiempo que aprovechan el ecosistema maduro y sólido de Spring para crear aplicaciones robustas y escalables. Esta combinación ganadora permite a los desarrolladores diseñar soluciones modernas y de alto rendimiento, preservando al mismo tiempo la compatibilidad con los proyectos existentes basados en Java. Con Kotlin, todo lo que se puede hacer con Java también es posible.

1.1 Beneficios

Kotlin ofrece varias ventajas sobre Java a la hora de desarrollar aplicaciones reactivas.

1.1.1 Código conciso

Kotlin es un lenguaje más conciso que Java, lo que significa que puede escribir menos código para realizar la misma tarea. Esto hace que el código sea más legible y mantenible, lo que es especialmente importante en aplicaciones reactivas donde la gestión de flujos y eventos puede ser compleja.

Por ejemplo, para crear un flujo reactivo de números del 1 al 10, en Java tenemos que escribir `Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`, mientras que en Kotlin es simplemente `(1..10).asFlow()`. Esta brevedad hace que el código sea más claro y fácil de leer.

```
// Java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squaredNumbers = numbers.stream()
                                      .map(n -> n * n)
                                      .collect(Collectors.toList());

// Kotlin
val numbers = listOf(1, 2, 3, 4, 5)
val squaredNumbers = numbers.map { it * it }
```

1.1.2 Valores null

En Kotlin, los valores `null` se manejan explícitamente. Esto reduce el riesgo de errores de tipo `NullPointerException`, que son un problema común en las aplicaciones reactivas.

En Kotlin, es posible declarar una variable de tipo `String` como nula añadiendo un signo de interrogación después del tipo (`String?`). Esto obliga al desarrollador a comprobar el estado de la variable antes de usarla, evitando así los `NullPointerException`.

```
// Java
String name = getName();
if (name != null) {
    // Hacer algo con name
}

// Kotlin
val name = getName()
name?.let {
    // Hacer algo con name
}
```

1.1.3 Extensiones de función

Kotlin permite definir extensiones de función, lo que permite la posibilidad de añadir funcionalidades a clases existentes sin tener que heredar de ellas. Esto facilita la adición de funcionalidad de reactividad a los tipos de datos existentes.

```
import reactor.core.publisher.Flux
// Extension de la clase Flux para añadir una función
// de filtro personalizada
fun Flux<Int>.filterEven(): Flux<Int> = this.filter { it % 2 == 0 }
fun main() {
    // Crear un Flux con números
    val numberFlux = Flux.just(1, 2, 3, 4, 5, 6)
    // Usar la extensión de la función filterEven
    val evenNumberFlux = numberFlux.filterEven()
    // Suscribirse para mostrar los elementos filtrados
    evenNumberFlux.subscribe { println(it) }
}
```

En este ejemplo, la función `filterEven` es una extensión de la clase `Flux` que sólo conserva los elementos pares del flujo. Utiliza el método `filter()` existente en `Flux` para realizar esta operación.

1.1.4 Smart cast

Kotlin ofrece conversiones inteligentes (*smart casts*), que permiten al compilador reconocer automáticamente el tipo de una variable en contextos específicos. Así, si se realizan verificaciones de tipo son realizados dentro de una variable, el compilador es capaz de convertir automáticamente esta variable dentro del tipo correspondiente a la verificación.

```
// Java
Object obj = getSomeObject();
if (obj instanceof String) {
    String str = (String) obj;
    // Haz algo con str
}

// Kotlin
val obj: Any? = getSomeObject()
if (obj is String) {
    // No hace falta cast, obj es considerado automáticamente
    // como un String aqui
    // Haz algo con obj
}
```

1.1.5 Coroutines

Kotlin ofrece soporte nativo para coroutines, que son un mecanismo que permite manejar operaciones asíncronas más fácilmente sin añadir complejidad al código. Por ejemplo, podemos usar coroutines para hacer llamadas asíncronas a la red sin tener que usar callbacks o flujos reactivos, lo que simplifica considerablemente el código.

```
// Kotlin
suspend fun fetchData(): String {
    return withContext(Dispatchers.IO) {
        // Efectuar una llamada asíncrona aquí
        // "Datos recuperados"
    }
}

// Utilización de la coroutine
val data = runBlocking { fetchData() }
```

1.1.6 Interoperabilidad con Java

Kotlin es totalmente interoperable con Java, lo que significa que puede utilizar las bibliotecas Java existentes dentro de sus aplicaciones Kotlin reactivas. Esto facilita la transición gradual a Kotlin de los proyectos existentes. Por ejemplo, puede utilizar bibliotecas reactivas Java como Reactor o RxJava en un proyecto Kotlin sin ningún problema.

```
// Java
Flux<String> flux = Flux.just("Hello", "World");
Mono<List<String>> mono = flux.collectList();

// Kotlin
val flux: Flux<String> = Flux.just("Hello", "World")
val mono: Mono<List<String>> = flux.collectList()
```

1.1.7 Legibilidad

Kotlin ofrece una sintaxis más expresiva y legible que Java. Esto hace que el código reactivo sea más fácil de entender, mantener y depurar.

```
// Java
List<String> fruits = Arrays.asList("Manzana", "Plátano", "Naranja");
List<String> upperCaseFruits = fruits.stream()
                                     .map(f -> f.toUpperCase())
                                     .collect(Collectors.toList());

// Kotlin
val fruits = listOf("Manzana", "Plátano", "Naranja")
val upperCaseFruits = fruits.map { it.toUpperCase() }
```

Por lo tanto, Kotlin ofrece una experiencia de desarrollo más agradable y productiva para las aplicaciones reactivas gracias a su concisión, la *null safety* extensiones de funciones, coroutines y compatibilidad con Java. Esto convierte a Kotlin en una opción atractiva para los desarrolladores que desean crear aplicaciones reactivas eficientes y escalables.

1.2 Exclusividades de Kotlin

Cualquier cosa que haga en Java se puede hacer en Kotlin. IntelliJ incluso ofrece traducciones sobre la marcha de Java a Kotlin utilizando copiar y pegar, con código copiado en Java seguido de código pegado en Kotlin. Java intenta ponerse al día con Kotlin a medida que se publica cada versión, pero siempre va un paso por detrás.

Kotlin es un lenguaje de programación moderno que ofrece una serie de características y mejoras con respecto a Java. Aquí tienes algunos ejemplos de lo que se puede hacer en Kotlin que no es posible en Java.

1.2.1 Funciones inlíne

Kotlin permite declarar funciones `inline` con la palabra clave `inline`, lo que mejora el rendimiento al evitar llamadas a funciones adicionales.

```
// Inline function
inline fun measureTime(block: () -> Unit) {
    val startTime = System.currentTimeMillis()
    block()
    val endTime = System.currentTimeMillis()
    println("Tiempo de ejecución: ${endTime - startTime} ms")
}
```

1.2.2 Inferencia de tipos

Kotlin tiene un sistema de inferencia de tipos más potente que Java, que permite declarar variables sin especificar explícitamente su tipo.

```
// Inferencia de tipos
val name = "John"
val age = 30

// No es necesario especificar explícitamente los tipos de
variables
val fruits = listOf("Apple", "Banana", "Orange")
val scores = mapOf("John" to 100, "Alice" to 90)
```

1.2.3 Interpolación de cadenas

Kotlin soporta la interpolación de cadenas, lo que facilita la concatenación de variables y literales en una cadena.

```
// Interpolación de cadenas
val name = "Juan"
val age = 30
// Concatenación de variables y literales en una caden
val mensaje = "Hola, mi nombre es $name y tengo $age años."
```

1.2.4 Excepciones checked

Kotlin no tiene excepciones `checked`, lo que significa que no tiene que manejar o declarar excepciones en un método. Esto hace que el código sea más conciso y menos verboso.

```
// Sin excepciones "comprobadas"
fun readFile() {
    try {
        // Código de lectura del archivo
    } catch (e: IOException) {
        // Manejo de excepciones (opcional)
    }
}
```

1.2.5 Expresión when

Kotlin proporciona una expresión `when` más potente que el tradicional `switch` de Java. `when` soporta múltiples tipos y condiciones más complejas.

```
// Expresión when
fun getMessage(status: Int): String {
    return when (status) {
        200 -> "OK"
        in 300..399 -> "Redireccion"
        404 -> "Not Found"
        else -> "Error desconocido"
    }
}

// Usando la expresión when
val status = 404
val message = getMessage(status)
```

Estas y otras características hacen de Kotlin un lenguaje más expresivo, conciso y seguro que Java, lo que lo hace muy atractivo para los desarrolladores modernos. Sin embargo, es importante tener en cuenta que Kotlin y Java son compatibles, lo que significa que puedes integrar código Kotlin en proyectos Java existentes y viceversa.

2. Uso de Kotlin con Reactor

Reactor soporta Kotlin 1.1+ y requiere `kotlin-stdlib` (o una de sus variantes `kotlin-stdlib-jdk7` y `kotlin-stdlib-jdk8`). Las extensiones Kotlin están en el módulo dedicado `reactor-kotlin-extensions` con un nombre de paquete que empieza por `reactor.kotlin`. La API KDoc de Reactor lista y documenta todas las extensiones Kotlin disponibles.

2.1 Null safety extension

Reactor va más allá proporcionando una null safety para toda su API utilizando anotaciones declaradas en el paquete `reactor.util.annotation`. Esto proporciona a los desarrolladores de Kotlin una seguridad nula completa, incluso con tipos Java utilizados en Kotlin. Las verificaciones de JSR 305 pueden configurarse para reforzar la seguridad nula según las necesidades del proyecto. Aunque algunas características aún no están totalmente soportadas, el equipo planea incluirlas en futuras versiones. Puede configurar las comprobaciones JSR 305 añadiendo el indicador del compilador `-Xjsr305` con las siguientes opciones: `-Xjsr305={strict|warn|ignore}`.

2.2 Tipos reificados

Los parámetros de tipo reificados en Kotlin son una funcionalidad que permite retener información sobre tipos genéricos en tiempo de ejecución. A diferencia de Java, donde los tipos genéricos se borran en tiempo de compilación, Kotlin permite retener información sobre estos tipos, facilitando su manipulación de forma segura y expresiva. Esto significa que puede acceder a la información sobre tipos genéricos, como el tipo real de una lista, en tiempo de ejecución. Esto hace que el código sea más robusto y permite detectar errores de tipado en tiempo de ejecución en lugar de en tiempo de compilación.

El borrado de tipos genéricos de la JVM es el proceso por el cual la información específica de los tipos genéricos se elimina durante la compilación en el código de bytes de Java, lo que significa que esta información no está disponible en tiempo de ejecución, haciendo que los tipos genéricos no estén reificados y sean potencialmente más difíciles de manipular en tiempo de ejecución.

Reactor proporciona extensiones para aprovechar esta funcionalidad:

Java	Kotlin con extensiones
`Mono.just("foo")`	`"foo".toMono()`
`Flux.fromIterable(list)`	`list.toFlux()`
`Mono.error(new RuntimeException())`	`RuntimeException() .toMono()`
`Flux.error(new RuntimeException())`	`RuntimeException().toFlux()`
`flux.ofType(Foo.class)`	`flux. ofType<Foo>() orflux.ofType(Foo::class)`
`StepVerifier.create (flux).verifyComplete()`	`flux.test().verifyComplete()`