Spring Annotations vs Kotlin DSL. Facciamo un benchmark

Perchè dovresti considerare di abbandonare le annotazioni a favore del DSL



Photo by Braden Collum on Unsplash

Con la versione 5.0 di Spring viene introdotto ufficialmente il DSL (Domain Specific Language) Kotlin per la definizione di bean e rotte.

Avete presente quella sensazione di incertezza quando introducete nuovi elementi nel sistema, collegandoli tramite annotazioni?

Vi è mai capitato di spendere ore di debug, solo per

accorgervi di avere messo il bean in un package che non è scansionato? O di avere sbagliato a valorizzare un attributo di qualche annotazione? O che state dimenticando qualche spring-boot-starter-something?

Dimenticatelo. **Se il DSL compila, è certo che spring sia in grado di risolvere i vostri bean**!

C'è un secondo grosso vantaggio scegliendo il DSL. Dalla documentazione di Spring:

> *His mechanism is very efficient, as it does not require any*
>
> *reflection or CGLIB proxies*

Fantastico! **Ma quanto efficiente**? Ho fatto un benchmark e con questo articolo vi presento i risultati.

Metodo di valutazione

Il codice sorgente del benchmark è qui.

Ho creato due moduli Maven:

- **nodsl:** un' applicatione sprinboot con classiche annotazioni.
- **dsl1**: un' applicazione springboot inizializzata con il DSL.

I bean nel modulo con le annotazioni hanno la seguente gerarchia di dipendenza:

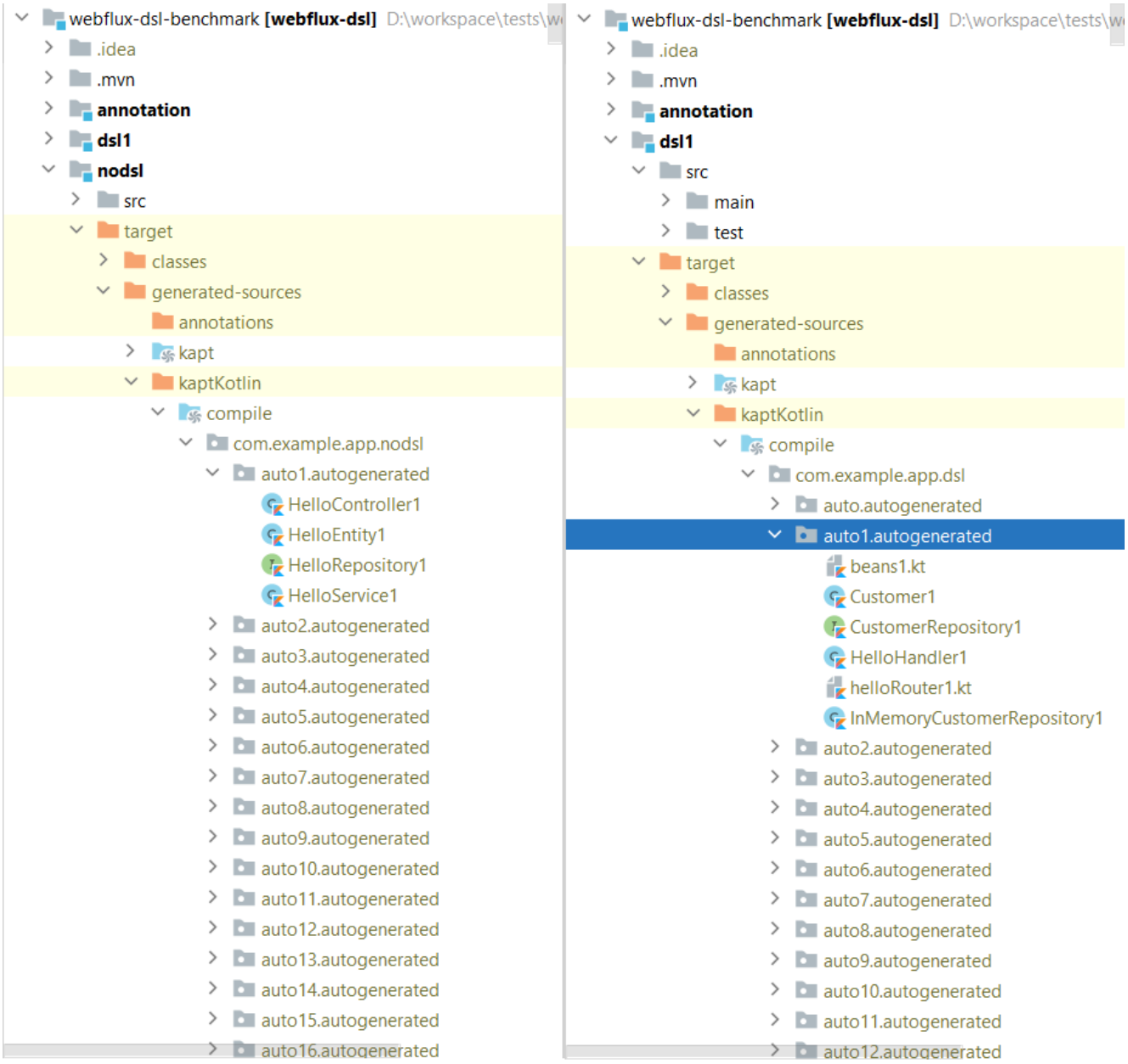
```
Controller
|__ Service |__ Repository (con JPA)
```

I bean del DSL, hanno invece quest'altra gerarchia:

```
Router
|__ Handler |__ Repository (senza JPA)
```

Ho creato un **annotation processor** usando Kapt per replicare

queste strutture. Ho eseguito il test con 1000 repliche di ognuna. Ogni replica ha un suo package dedicato.



La struttura autogenerata dei package

La gerarchia dei Controller è registrata nello ***Spring context*** tramite le tipiche annotazioni @RestController, @Service, @Repository, @Entity, @Component.

La gerarchia dei Router è invece registrata come segue:

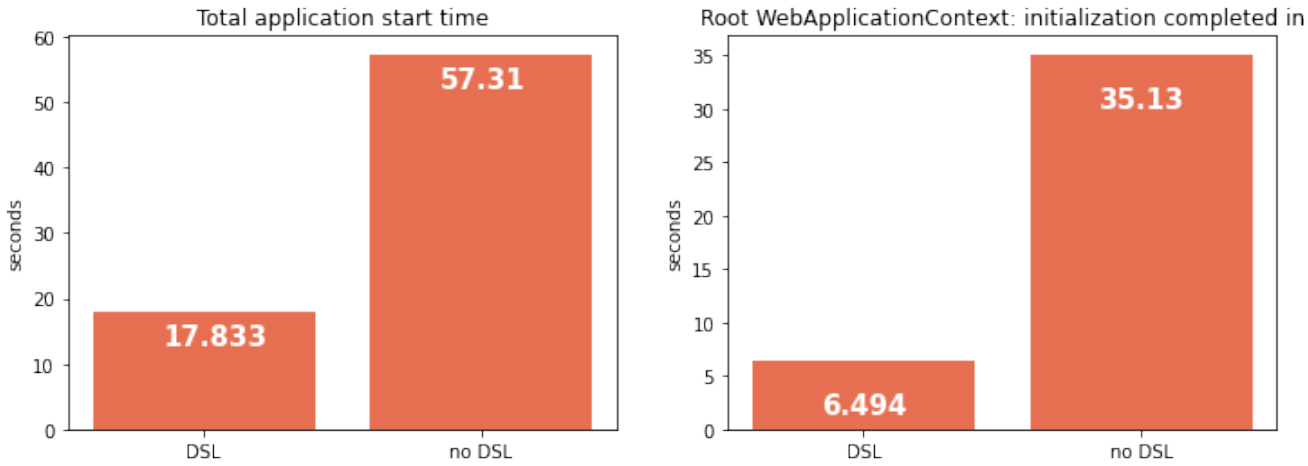
```
@HowManyRoutes(1000)
@SpringBootApplication(scanBasePackages = [])
class DslApplication fun main(args: Array<String>) {
runApplication<DslApplication>(*args){
```

```
addInitializers(allBeans()) }
}
fun allBeans(): BeanDefinitionDsl { return beans {
beans1(this) beans2(this) ... beans1000(this) }
}
public fun beans1(dsl: BeanDefinitionDsl): Unit {
dsl.bean<HelloHandler1>()
dsl.bean<InMemoryCustomerRepository1>()
dsl.bean(::helloRouter1)
}
public fun helloRouter1(handler: HelloHandler1) = router {
"hello1".nest { GET("", handler::hello) }
}
```

Risultati

I log dell'avvio dell' applicazione sono nella root del repository:

Segue un grafico dei risultati del primo avvio.



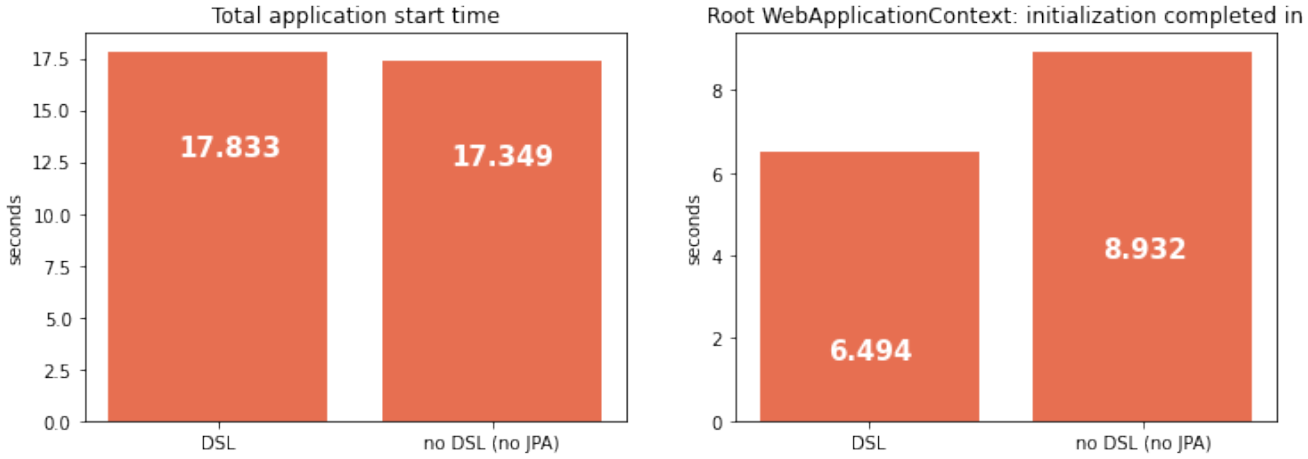
Risultati del primo avvio

A prima vista sembrano molto promettenti. La differenza è significativa sia nel tempo di inizializzazione del contesto che in quello di avvio completo.

Devo però precisare che nel caso del DSL **non ho utilizzato JPA**. Come si vede dai log, lo scan dei repository impiega 22.185 secondi, nel caso "no DSL". Non ho attivato la creazione automatica delle tabelle (spring.jpa.hibernate.ddl-

auto=none).

Quindi cosa succede **se rimuovo JPA**, mantenendo però la stessa gerarchia di dipendenze? Ho sostituito @Repository con un normale @Component e rimosso dal pom lo starter JPA spring-boot-starter-data-jpa. Il codice di questa prova è nel branch no-jpa del repo Github.



Risultati del benchmark senza JPA

Il risultato mi ha sorpreso parecchio. **Praticamente nessuna differenza** tra DSL e annotazioni, se escludiamo l'utilizzo di JPA. Solo una piccola differenza nel tempo di inizializzazione del contesto, ma il tempo totale di avvio è praticamente lo stesso.

Discussione



Photo by Icons8 Team on Unsplash

Ho iniziato questo benchmark chiedendomi quanto l'utilizzo del DSL **sarebbe stato utile nel migliorare i tempi di avvio** di un' applicazione Springboot. Dopo il benchmark posso affermare che **non c'è alcun beneficio diretto nel DSL in sè**.

JPA fa uso di generazione di codice tramite CGLIB. Questo benchmark è prova di quanto il suo utilizzo impatti sui tempi di avvio dell'applicazione. Anche l'annotazione @Transanctional ha l'effetto di wrappare il bean in un proxy CGLIB autogenerato.

Ne deduco che se si vuole utilizzare Springboot mantenendo dei tempi di avvio accettabili, è utile **evitare le casistiche in cui Spring crea dei proxy CGLIB**. Che significa evitare JPA, l' annotazione @Transactional e anche i @Bean factory method dichiarati in una classe annotata come @Configuration.

Onestamente, JPA non mi piaceva nemmeno prima di questo benchmark. Adesso ho una ragione in più per evitarlo.

Ho fatto qualche prova con Exposed, e lo utilizzerò certamente come sostituto di JPA nei prossimi progetti. Exposed va nella stessa direzione del DSL Spring, ovvero evitare la “Convention over configuration”. E’ un progetto giovane, ma è utilizzato da 6 anni in produzione da JetBrains, quindi si può considerare stabile.

Conclusioni

Non c’è un beneficio diretto nell’utilizzo del DSL, in termini di scansione e injection dei bean. Tuttavia, la premessa era che le performance di boot sarebbero migliorate grazie al **mancato utilizzo di CGLIB e della reflection**.

Con questo benchmark ho dato prova di quanto sia grave l’impatto di CGLIB. Quindi, utilizzare un *coding style* che ci sproni e supporti nel evitare CGLIB, ha il beneficio indiretto di migliorare i tempi di boot.

“Convention over configuration” ha un appeal quasi magico quando lo si approccia le prime volte. Sfortunatamente **può però tramutarsi molto velocemente in una maledizione**, quando si tratta di debuggare situazioni complesse.

Uno stile più dichiarativo, aiutato dal compilatore, **può salvarci da ore di debug**. Affidarci al supporto di un compilatore, ci permette di essere più confidenti in ciò che scriviamo. Inoltre, riduce la probabilità di utilizzare un tool nel modo sbagliato.

Nei miei prossimi progetti, non ho dubbi che opterò per la definizione dei bean tramite DSL ed eviterò JPA e Hibernate.

E a voi come suona l’idea di abbandonare JPA, Hibernate e la “Convention over Configuration”? Vi sembra una pazzia o non vedete l’ ora di salutarle?

Grazie per la lettura!

https://italiancoders.it/spring-annotations-vs-kotlin-dsl-facciamo-un-benchmark/