

*Spring Framework* è essenzialmente un prodotto capace di facilitare, snellire e velocizzare lo sviluppo di applicazioni web (e non) basate su Java.

Con la introduzione della *Dependency Injection* *Spring Framework* permette di sviluppare applicazioni completamente disaccoppiate dalla creazione degli oggetti (delegata al framework mediante Reflection) che è *customizzabile* mediante i soli files di configurazione. La filosofia di facilitare il lavoro di sviluppo è tipica del mondo Spring.

Ecco alcuni esempi significativi.

1) La gestione sofisticata delle transazioni in Spring, ad esempio, sarebbe un vero rompicapo se implementata completamente da zero dallo sviluppatore.

2) Il modulo *spring-data*, tra le tante altre cose, mette a disposizione dello sviluppo un database *embedded* ed alcune interfacce, quali ad esempio la *JpaRepository* che è sufficiente estendere per usufruire di ben 18 metodi di interfaccia senza implementarli. Si preoccuperà Spring a *runtime* di creare le classi concrete che implementano detti metodi. Si possono inoltre aggiungere metodi a questa interfaccia mediante un metalinguaggio convenzionale e verranno anch'essi concretamente implementati a *runtime*. Questo svincola lo sviluppatore dall'implementare e testare un DAO generico come si faceva in passato (o ciò che è peggio NDAO!!!).

3) Un altro esempio è *spring-security* che, fra le tante altre cose, aggiunge ad esempio il *token* delle richieste nelle viste per prevenire la CSRF.

Gli esempi sarebbero centinaia e presupporrebbero trattare Spring nel profondo, ma questi esempi sono sufficienti a capirne l'estrema utilità.

Inoltre il Framework mantiene un aggiornamento costante rispetto ai moderni sistemi di sviluppo di applicazioni Web disponendo, ad esempio, di moduli per la programmazione sofisticata di architetture a

Microservizi (con artefatti che implementano gli Enterprise Integration Patterns) o per lo sviluppo di Applicazioni Web reattive, o per l'integrazione con Social Network, etc.

*Spring Boot* rappresenta a sua volta uno strumento capace di rendere estremamente semplice

l'uso di *Spring Framework* mediante strumenti per la build quali *Maven* o *Gradle*.

L'uso di Spring infatti, ma in generale di un applicativo composto da tante dipendenze, è spesso osteggiato dal problema di dover far coincidere le versioni dei vari moduli (o artefatti) componenti il

progetto in fase di *build*, o dalla verbosità di configurazioni che quasi sempre si ripetono (esempio il file

*web.xml* di una *web application* con Spring MVC). Configurazioni che pertanto in *Spring Boot*

sono divenute **convenzioni** eventualmente *customizzabili*, difatti un progetto *Spring Boot*

si autoconfigura.

Quindi

*Spring Framework* + **convenzioni** + *Maven* (o *Gradle*) = *Spring Boot*.



Sostanzialmente prima di *Spring Boot* per creare un progetto contenente i vari moduli di

Spring (e non) bisognava conoscere esattamente quali artefatti fossero compatibili con gli altri, quali fossero le reali dipendenze di un artefatto col rischio di dimenticarne qualcuna accorgendosene solo a

runtime. Quindi per gestire ad esempio la persistenza bisognava includere il modulo *spring-data*, il

modulo delle specifiche JPA, i moduli del framework ORM che implementa JPA, poi l'artefatto delle

specifiche JDBC, i moduli del vendor JDBC, etc. ed ogni modulo doveva essere compatibile con gli altri in

termini quindi di versioni, inoltre in fase di *build* c'era il rischio di dimenticare qualche modulo e di

scoprirlo solo a runtime. Con *Spring Boot* invece tutti questi moduli sono stati accorpati in

famiglie di moduli ad esempio nell'artefatto *spring-boot-starter-data-jpa*

(a cui va aggiunto il modulo dei connettori cioè i moduli del vendor JDBC) così come sono stati accorpati i moduli della parte web in *spring-boot-starter-web*

oppure il *security-starter* per quanto riguarda tutte le dipendenze del comparto sicurezza.

Nascono quindi gli **starter** che sono degli artefatti composti fra loro compatibili, e quindi sono un

insieme di dipendenze *Maven*, che riguardano un preciso contesto architetturale. *Spring*

*Boot* coordina ovviamente anche la compatibilità fra gli starter mediante il centro di coordinamento è

l'artefatto *spring-boot-starter-parent* di cui tutti i progetti *Spring Boot* sono figli.

E' infatti importante sottolineare che in un progetto *Spring Boot* vi è una sola versione da

specificare: **quella del pom padre**. Le sofisticazioni di *Spring Boot* sono infatti

essenzialmente dovute all'uso dell'ereditarietà in *Maven*. Mediante *Maven* quindi *Spring*

*Boot* si preoccupa anche del fatto di evitare che vi siano dipendenze duplicate, e quindi classi

duplicate (evita quindi rompicapi a runtime), scegliendo sempre la versione più recente di un determinato

artefatto qualora transitivamente vi siano più dipendenze dello stesso.

Per quanto riguarda l'autoconfigurazione invece, come esempio, è utile sottolineare nelle applicazioni Spring Web la complessità dell'inserimento di tutte le necessarie dipendenze, o la verbosità del file `web.xml` (con, fra le altre cose, la creazione della `DispatcherServlet` e beans annessi), l'abilitazione di Spring Web MVC nella configurazione, e la necessità di un Servlet Container. Tutti questi passaggi vengono superati da **Spring Boot** banalmente inserendo lo starter web che automaticamente:

1) provvede a configurare mediante convenzioni tutti gli oggetti creati in un ipotetico file `web.xml` (che non c'è), quindi a creare `DispatcherServlet` con i beans annessi e poi `Filter` e `ServletContextInitializer`.  
Tutti eventualmente *customizzabili*.

2) include inoltre un Servlet Container.

In merito a questo c'è da sottolineare una cosa importante:

a differenza di una classica applicazione web in cui si lancia il server (il *main method* è nel server) e si

*deploya* su di esso un WAR (prodotto dello sviluppo), un'applicazione web implementata via **Spring**

**Boot** è invece un JAR caratterizzato da una classe con un *main method* che rappresenta il boot

dell'applicativo (annotata con `@SpringBootApplication` ed è l'*entry point* di **Spring Boot**) la

quale lancia Spring poi lancia il Servlet Container (*embedded*) sul quale poi **Spring Boot** stesso

fa il *deploy* della *web application*. Nel caso quindi di una classica applicazione **Spring Boot** in

cui dovesse servire il WAR (e non si abbia quindi scelto l'opzione war nell'*Initializer*) relativo

all'applicazione per *deployarlo* su altro server, o comunque per avere un WAR server-independent, vi sono alcuni passaggi da seguire:

1) cambiare ovviamente il packaging nel pom in `<packaging>war</packaging>`

Il war generato da **Maven** non sarebbe *deployabile* in quanto l'applicativo necessita comunque di un

`web.xml` per essere utilizzabile, e quindi serve inizializzare e configurare la `DispatcherServlet` con i

beans annessi e poi `Filter` e `ServletContextInitializer`.

Con la specifica *Servlet 3.0* ogni Servlet Container va alla ricerca di un oggetto di tipo

[SpringServletContainerInitializer](#) il quale parte alla ricerca di oggetti che implementano [WebApplicationInitializer](#) e nel modulo spring-web si trova appunto la classe astratta [SpringBootServletInitializer](#).

Quindi per *completare il giro* basta concretizzare detta classe ,ad esempio in questo modo:

```
2)public class CustomServletInitializer
extends SpringBootServletInitializer {
@Override
protected SpringApplicationBuilder configure(
SpringApplicationBuilder builder) {

return builder.sources(ApplicationWithMain.class);
}
}
```

... dove [ApplicationWithMain](#) è appunto la main class di *Spring Boot* , la quale ha ora un

duplice funzione: fungere da starter qualora qualora si volesse tornare al .JAR, e da web.xml nel

.WAR, inoltre qualora vi fossero altri files di configurazione l'annotazione [@SpringBootApplication](#) abilita il component scanner quindi non vi è neanche la necessità di aggiungerli via codice allo SpringApplicationBuilder in quanto il Framework li troverà automaticamente.

Dal sito <https://start.spring.io> si può costruire, inserendo le dipendenze di cui si necessita, in modo semplice ed intuitivo un progetto *Spring Boot* e lavorarci in qualsiasi IDE

importandolo(dopo averlo scompattato) come **Maven** Project. Il file .zip risultato è detto **Initializr**. Ovviamente in vari IDE hanno un wizard per comporre un **Initializr** simile a quello del sito web poiché

puntano ad esso. Il file .zip contiene, fra le altre cose, anche il job batch Windows che eventualmente scarica e configura **Maven**.

Definirei quindi *Spring Boot* banalmente un facilitatore dell'uso dello *Spring*

*Framework* senza il quale *Spring Boot* non avrebbe ragione di esistenza, quindi

*Spring Boot* non aggiunge moduli(o codice) a Spring ma li coordina, non è un Application

Server in quanto una applicazione *Spring Boot* può anche non avere la parte Web col server

annesso, non è una alternativa(né può essere considerato un concorrente)a Spring in quanto è composto da moduli Spring.

Programmare pertanto direttamente con *Spring Boot* però(ad esempio senza aver mai usato o

configurato *Spring Framework*), potrebbe rendere lo sviluppo estremamente vulnerabile

se si ignora quello che avviene *behind the scenes*....