Luca Cabibbo

Architettura dei Sistemi Software

Introduzione a Spring

dispensa asw820

ottobre 2023

You can cut all the flowers
but you cannot keep Spring from coming.

Pablo Neruda

- Riferimenti

- Spring website
  - https://spring.io/
- **Spring Framework Reference Documentation**, 6.0.12, 2023.
  - https://docs.spring.io/spring-framework/docs/current/reference/html/
- Walls, C. **Spring in Action**, sixth edition. Manning, 2022.
- Spilcă, L. **Spring Start Here**, Manning, 2021.

- Obiettivi e argomenti

- Obiettivi
  - fornire una breve introduzione al framework Spring
  - presentare l'iniezione delle dipendenze in Spring
  - introdurre brevemente la gestione delle dipendenze
- Argomenti
  - introduzione
  - introduzione al framework Spring
  - bean e iniezione delle dipendenze
  - gestione delle dipendenze
  - discussione

* Introduzione

- Le esercitazioni di questo corso enfatizzano i microservizi con Spring (Spring Boot e Spring Cloud), Docker e Kubernetes
  - Spring è un framework applicativo modulare, per lo sviluppo di applicazioni Java di tipo enterprise
  - Spring Boot fornisce un ulteriore supporto e semplificazione nello sviluppo di applicazioni basate su Spring
  - Spring Cloud fornisce poi il supporto per lo sviluppo di applicazioni distribuite per il cloud
  - questa dispensa presenta alcune idee di base del framework Spring, utili per la comprensione e l'utilizzo di Spring Boot e Spring Cloud

* Introduzione a Spring

- *Spring Framework* (o semplicemente *Spring*) è un framework applicativo, open source, che ha l'obiettivo semplificare lo sviluppo di applicazioni Java di tipo enterprise
  - Spring è stato inizialmente creato nel 2002, come approccio in qualche modo alternativo alla piattaforma Java EE (allora chiamata J2EE) – per raggiungere scopi analoghi ma con un modello di programmazione più semplice e più leggero
  - nel corso del tempo, Spring è evoluto in modo significativo – in termini di modello di programmazione, di funzionalità, nonché di supporto per nuove tecnologie
  - nel frattempo anche la piattaforma Java/Java EE è evoluta in modo significativo – talvolta facendo proprie anche alcune idee innovative di Spring
  - oggi Spring è un framework modulare ma coeso, in grado di supportare una varietà di necessità applicative – come le applicazioni web, la sicurezza, i microservizi e il cloud

Strategie adottate da Spring

- Il Framework Spring adotta diverse strategie per semplificare lo sviluppo di applicazioni Java
  - sviluppo leggero basato su POJO (Plain Old Java Object) – chiamati in Spring "bean"
  - accoppiamento debole basato sull'iniezione delle dipendenze (DI, Dependency Injection), insieme all'uso estensivo di interfacce
  - programmazione dichiarativa basata su aspetti, configurazioni e convenzioni comuni
  - eliminazione di codice ripetuto ("boilerplate") mediante aspetti e template

Bean e POJO

- Sviluppo leggero basato su POJO (Plain Old Java Object)
  - alcuni framework obbligano a “sporcare” il codice applicativo con l’uso delle proprie API specifiche – scrivendo classi che estendono classi o che implementano interfacce di queste API
  - al contrario, Spring (per quanto possibile) evita che il codice applicativo sia accoppiato all’uso delle proprie API
  - il modello applicativo di Spring è basato su semplici oggetti o classi *POJO* (*Plain Old Java Object*) detti anche *bean*
    - in teoria, un POJO è una classe che non segue nessun modello o convenzione o framework specifico – in particolare, non estende nessuna classe predefinita, non implementa nessuna interfaccia predefinita, non contiene nessuna annotazione predefinita
    - in pratica, i bean di Spring violano questa definizione in vari modi – ad es., è comune l’uso delle convenzioni dei metodi get/set (JavaBean) e di alcune annotazioni specifiche

Iniezione delle dipendenze

- Accoppiamento debole basato sull’iniezione delle dipendenze
  - le applicazioni non banali sono composte da più classi e oggetti, che collaborano tra loro mediante messaggi/invocazioni
  - gli oggetti dell’applicazione hanno *dipendenze* verso altri oggetti – infatti ciascun oggetto deve conoscere (i riferimenti a) gli altri oggetti a cui deve inviare messaggi per collaborare
  - un progetto in cui gli oggetti sono responsabili di acquisire i riferimenti agli oggetti da cui dipendono (o addirittura anche di crearli) può essere altamente accoppiato e difficile da testare
  - l’*iniezione delle dipendenze* (*Dependency Injection*, *DI*) è un meccanismo per assegnare (iniettare) agli oggetti le loro dipendenze, al momento della creazione – gli oggetti non devono più occuparsi di acquisire direttamente le proprie dipendenze – questo riduce l’accoppiamento, e inoltre favorisce il testing

Configurazioni e convenzioni comuni

- Programmazione dichiarativa basata su configurazioni e convenzioni comuni
  - in Spring, molte attività (come l'iniezione delle dipendenze) sono basate sulla *specifica dichiarativa* delle configurazioni delle classi e degli oggetti – anziché sulla scrittura di codice imperativo
  - i metadati di configurazione possono essere specificati come configurazioni basate su XML, configurazioni basate su Java (mediante l'uso di annotazioni) e file di proprietà
  - Spring fornisce inoltre dei meccanismi di configurazione automatica e implicita, anche basata sull'adozione di valori di default e convenzioni comuni – questi meccanismi sono poi estesi in Spring Boot

Template

- Eliminazione di codice ripetuto mediante template
  - molte tecnologie (e le loro API) richiedono di scrivere (più volte) una gran quantità di codice, pieno di dettagli, anche per svolgere delle attività semplici e comuni – il cosiddetto "boilerplate code" ("blocchi di codice standard")
    - ad es., si pensi al codice necessario per eseguire un'interrogazione SQL con JDBC e poi ricostruire un oggetto a partire dal risultato dell'interrogazione
  - Spring fornisce, per diverse tecnologie, delle classi di utilità (chiamate *template*) per svolgere le attività più comuni in modo semplificato, eliminando la necessità di "boilerplate code" e dunque riducendo la quantità di codice da scrivere (e da testare e da mantenere)

* Bean e iniezione delle dipendenze

- Ogni applicazione non banale comprende diversi oggetti o componenti – ciascuno responsabile di una propria parte delle funzionalità dell'applicazione – che devono interagire e collaborare
  - quando l'applicazione viene eseguita, questi componenti devono essere creati e collegati tra loro
    - questo può essere gestito direttamente dai componenti stessi o mediante oggetti di supporto (ad es., factory) – oppure con l'ausilio di un framework
  - Spring, nel suo nucleo, offre un container (o application context) proprio per creare e gestire i componenti di un'applicazione (chiamati bean)
    - il collegamento tra i componenti avviene mediante un meccanismo di iniezione delle dipendenze

Iniezione delle dipendenze

- L'iniezione delle dipendenze – *Dependency Injection* (*DI*) – è una caratteristica fondamentale del framework Spring
  - un'applicazione Spring è tipicamente composta da molti oggetti (chiamati *bean*), che devono collaborare e, per questo, devono essere creati e collegati – perché questi bean hanno delle *dipendenze* tra di loro
  - per organizzare e comporre i bean in un'applicazione coerente, il framework Spring si occupa della gestione delle dipendenze mediante l'iniezione delle dipendenze
    - si parla anche di "inversione del controllo" o *Inversion of Control* (*IoC*), perché non sono i bean a dover gestire *direttamente* le loro dipendenze, ma piuttosto queste dipendenze vengono gestite da un'entità separata: il "contenitore Spring"

Contenitore Spring

- In un'applicazione Spring, i bean vivono in un *contenitore* (*container*) Spring – chiamato anche *Inversion of Control container* o *IoC container* – la nozione di contenitore è centrale nel framework Spring (e nelle tecnologie a componenti)
  - il contenitore ha la responsabilità di creare i bean, configurarli e collegarli tra di loro (mediante iniezione delle dipendenze) – e più in generale di gestire questi bean e il loro ciclo di vita (dalla creazione alla distruzione)
  - a tal fine, bisogna fornire una configurazione opportuna al contenitore Spring, che specifica quali bean deve creare, e come configurarli e collegarli tra di loro
    - i metadati di configurazione possono essere specificati sia in XML che mediante annotazioni Java oppure file di proprietà
  - l'interfaccia di programmazione (API) usata per interagire con il contenitore IoC di Spring è chiamata *application context*
    - di questa interfaccia esistono diverse implementazioni

Bean

- In Spring, gli oggetti gestiti dal contenitore sono chiamati *bean*
  - ciascun bean è un oggetto, a cui è assegnato un nome
  - ogni bean può avere delle dipendenze, verso altri bean
  - ci sono due meccanismi principali attraverso cui può avvenire l'iniezione delle dipendenze in un bean – tramite il costruttore (e i suoi parametri) e tramite i metodi set
    - in entrambi i casi è il contenitore che invocherà il costruttore e/o i metodi set
  - in effetti, è anche possibile l'iniezione delle dipendenze direttamente nei campi (variabili d'istanza) del bean – mediante meccanismi di riflessione

Bean

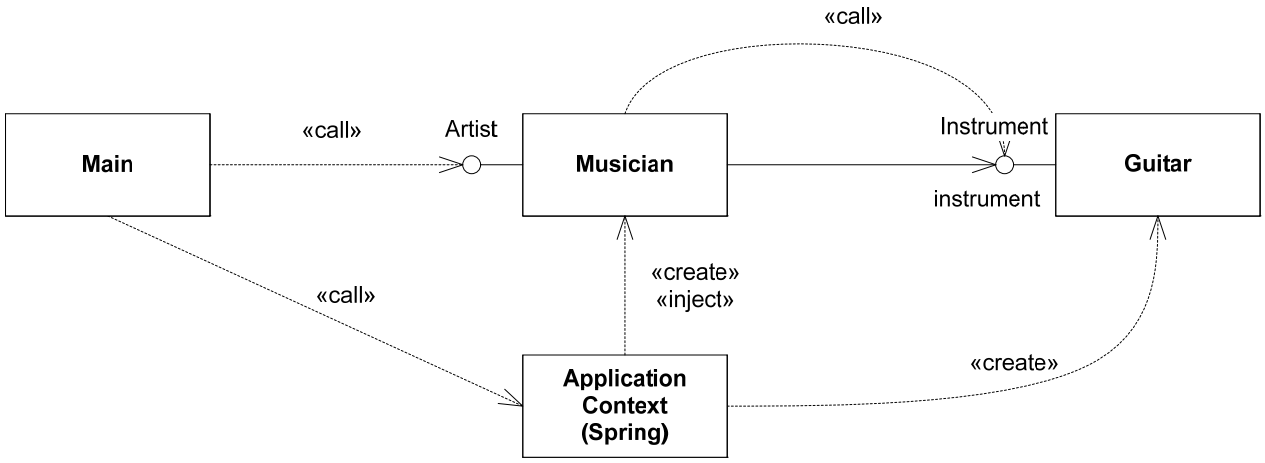
- In Spring, gli oggetti gestiti dal contenitore sono chiamati *bean*
  - in Spring, la nozione di “bean” è diversa sia da quella di “oggetto” che di “classe”
    - intuitivamente, un “bean” è una “tipologia di oggetti”
    - la definizione di un bean richiede una classe – ma ci possono essere più bean definiti da una stessa classe
    - ci possono anche essere più istanze di uno stesso bean
    - ogni bean è un oggetto, ma non tutti gli oggetti sono bean
      - alcuni o molti oggetti possono vivere fuori dal contenitore
    - i bean sono gli oggetti che vogliamo che vengano gestiti dal contenitore
      - ad es., servizi, repository, controller, …

- Configurazione basata su XML

- Si supponga di dover realizzare un’applicazione con un bean **hendrix** (che è un **Musician** che è un **Artist**) collegato a un bean **stratocaster** (che è una **Guitar** che è un **Instrument**)

  - la configurazione di questi bean (compreso il loro collegamento) può essere specificata mediante un file di configurazione XML
  - l’accesso a questi bean può poi avvenire mediante un application context – nell’esempio, di tipo **ClassPathXmlApplicationContext**, che legge la configurazione da un file XML

Esempio



Artist e Instrument

- Ecco la definizione delle interfacce **Artist** e **Instrument**

```
package asw.spring.show;

/* Un artista. */
public interface Artist {

    /* Esibizione dell'artista. */
    public String perform();

}
```

```
package asw.spring.show;

/* Uno strumento musicale. */
public interface Instrument {

    /* Suona lo strumento. */
    public String play();

}
```

Musician

- Ecco una possibile implementazione **Musician** di **Artist**

```
package asw.spring.show;

/* Un musicista (suona uno strumento). */
public class Musician implements Artist {

    /* il nome del musicista */
    private String name;
    /* lo strumento suonato dal musicista */
    private Instrument instrument;

    /* Crea un nuovo musicista. */
    public Musician(String name, Instrument instrument) {
        this.name = name;    this.instrument = instrument;
    }

    /* Esibizione del musicista. */
    public String perform() {
        return "I'm " + name + ": " + instrument.play();
    }

}
```


Guitar

- Ecco una possibile implementazione **Guitar** di **Instrument**

```
package asw.spring.show;

/* Una chitarra. */
public class Guitar implements Instrument {

    /* il suono della chitarra */
    private String sound;

    /* Crea una nuova chitarra. */
    public Guitar() { }

    /* Assegna il suono della chitarra. */
    public void setSound(String sound) {
        this.sound = sound;
    }

    /* Suona la chitarra. */
    public String play() {
        return sound;
    }

}
```

Configurazione XML

- Ecco il file di configurazione XML show-beans.xml
  - descrive i bean **hendrix** e **stratocaster** dell'applicazione – con i loro nomi e i loro tipi (classi) – nonché le loro dipendenze

```
<?xml version="1.0" encoding="UTF-8"?>
<beans …>

  <bean id="hendrix" class="asw.spring.show.Musician">
    <constructor-arg value="Jimi"/>
    <constructor-arg ref="stratocaster"/>
  </bean>

  <bean id="stratocaster" class="asw.spring.show.Guitar">
    <property name="sound" value="Ua ua uaa"/>
  </bean>

</beans>
```

nome (id) di un bean

classe del bean

un bean

una dipendenza da soddisfare

riferimento a un altro bean (tramite il nome)

La classe Main

- La classe **Main** ha bisogno di un riferimento al bean **hendrix**

```
package asw.spring.show;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
           ClassPathXmlApplicationContext;

public class Main {

   public static void main(String[] args) {
      ApplicationContext context =
          new ClassPathXmlApplicationContext("show-beans.xml");
      Artist hendrix = (Artist) context.getBean("hendrix");
      System.out.println( hendrix.perform() );
   }

}
```

  - l'esecuzione di questa applicazione visualizzerà la stringa I'm Jimi: Ua ua uaa

- Processo di risoluzione delle dipendenze

- Ecco come viene gestita la risoluzione delle dipendenze dal contenitore IoC di Spring
  - il contenitore viene inizializzato con i metadati di configurazione – che vengono letti mediante l'application context
  - le dipendenze di ciascun bean sono costituite dagli argomenti del costruttore e dalle sue proprietà (che possono essere assegnate mediante metodi set)
    - il contenitore, al momento della creazione di ciascun bean, inietterà nel bean le sue dipendenze
    - il momento della creazione di un bean dipende dal suo scope (discusso dopo) – ad es., all'inizializzazione del contenitore oppure su richiesta
  - quando c'è bisogno di creare un bean, il contenitore determina, per ogni argomento del costruttore e per ogni proprietà, il riferimento a un altro bean nel contenitore (ref) oppure il valore (value, un letterale o costante) da assegnare/iniettare

Portata (scope) nella creazione dei bean

- Ciascun tipo di bean è caratterizzato da uno *scope* (portata), che definisce quanti bean creare e quando (per quel tipo di bean) – lo scope può essere specificato mediante l'attributo XML scope
  - ecco gli scope principali per i bean
    - *singleton* (è il default) – esattamente un'istanza del bean per contenitore – il bean viene creato di solito al momento dell'inizializzazione del contenitore
    - *prototype* – un'istanza per ciascuna richiesta di un bean di quel tipo
    - *request* – in un'applicazione web, un'istanza per ciascuna richiesta HTTP
    - *session* – in un'applicazione web, un'istanza per ciascuna sessione HTTP
    - *application* – in un'applicazione web, un'istanza per l'intera applicazione web

Discussione

- Nella configurazione basata su XML
  - le interfacce e le classi per i bean sono effettivamente dei POJO
  - i metadati di configurazione sono codificati in XML
  - il client di un bean può accedere al bean mediante l'application context, sulla base del nome (oppure del tipo) del bean di interesse

- Una diversa configurazione

- Usando le stesse classi, è possibile definire dei bean differenti – mediante una configurazione diversa
  - ecco i bean **may** e **redspecial**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans …>

  <bean id="may" class="asw.spring.show.Musician">
    <constructor-arg value="Brian"/>
    <constructor-arg ref="redspecial"/>
  </bean>

  <bean id="redspecial" class="asw.spring.show.Guitar">
    <property name="sound" value="La la laa"/>
  </bean>

</beans>
```

  - questi bean possono anche convivere con i precedenti bean

- Parentesi: annotazioni

- Le *annotazioni* (in Java e altri linguaggi di programmazione) sono un elemento sintattico utilizzato per annotare (etichettare) degli elementi di codice – come classi, interfacce, variabili o metodi
  - ad es., **@Test** (di JUnit), **@Bean** o **@Configuration** (di Spring)
  - il compilatore Java riporta le annotazioni nel bytecode come metadati – ma il compilatore non interpreta ulteriormente le annotazioni
  - le annotazioni sono prese in considerazione da strumenti di sviluppo opportuni (ad es., JUnit) e/o dall'ambiente di esecuzione (ad es., il framework Spring), che possono agire di conseguenza
    - ogni strumento/ambiente sa interpretare solo le annotazioni di propria competenza – ma ignora tutte le altre
  - nel seguito verranno esemplificate alcune annotazione del framework Spring

- Configurazione basata su Java

- Un approccio più moderno per specificare i metadati di configurazione di un'applicazione è mediante una configurazione basata su Java, tramite l'utilizzo di apposite annotazioni
  - una configurazione basata su Java richiede una classe di configurazione, che va annotata **@Configuration**, che definisce un metodo per ciascun bean (o tipo di bean)
  - il metodo per un bean deve essere annotato **@Bean** – il metodo deve creare il bean e collegarlo con altri bean (i cui riferimenti vanno ottenuti invocando i corrispondenti metodi) – va notato che sarà poi il contenitore a decidere quando effettivamente invocare questi metodi (il che non è ovvio)
  - l'accesso ai bean può avvenire mediante un application context di tipo **AnnotationConfigApplicationContext** – che ottiene la configurazione dalla classe di configurazione
  - nell'esempio (che è equivalente a quello di prima), le interfacce e le classi per i bean sono definite come prima

Classe di configurazione

- La classe di configurazione **ShowConfig**

```
package asw.spring.show.config;

import asw.spring.show.*;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;

/* Configurazione Spring per l'applicazione Show. */
@Configuration
public class ShowConfig {
    @Bean
    public Artist hendrix() {
        return new Musician("Jimi", stratocaster());
    }
    @Bean
    public Instrument stratocaster() {
        Guitar stratocaster = new Guitar();
        stratocaster.setSound("Ua ua uaa");
        return stratocaster;
    }
}
```

nome del bean

definisce un bean il cui nome è implicitamente "hendrix"

riferimento a un altro bean (tramite il metodo)

L'annotazione @Bean

- L'annotazione **@Bean** definisce un bean
  - è analogo a un elemento XML <bean/>
  - il nome del bean corrisponde implicitamente al nome del metodo
    - altrimenti può essere definito esplicitamente con **@Bean(name="hendrix")**
  - lo scope di un bean può essere specificato con l'annotazione **@Scope**

La classe Main

- La nuova classe **Main**

```
package asw.spring.show;

import asw.spring.show.config.ShowConfig;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.
           AnnotationConfigApplicationContext;

/* Applicazione che ottiene e avvia il client. */
public class Main {

   /* Ottiene e avvia un oggetto Client. */
   public static void main(String[] args) {
      ApplicationContext context =
          new AnnotationConfigApplicationContext(ShowConfig.class);
      Artist hendrix = (Artist) context.getBean("hendrix");
      System.out.println( hendrix.perform() );
   }

}
```


Discussione

- Nella configurazione basata su Java
  - le interfacce e le classi per i bean sono ancora effettivamente dei POJO
  - i metadati di configurazione sono codificati in Java, sotto forma di annotazioni
  - il client di un bean può ancora accedere al bean mediante l'application context, sulla base del nome (oppure del tipo) del bean di interesse

Confronto

- La configurazione basata su XML e su Java sono due modalità differenti per descrivere i metadati di configurazione necessari in un'applicazione Spring
  - il contenitore Spring è completamente indipendente dal modo in cui i metadati di configurazione vengono descritti
  - la configurazione basata su Java è più comune
    - ha alcuni piccoli vantaggi sulla configurazione basata su XML – ad es., maggior tipizzazione e refactorability – ma anche alcuni piccoli svantaggi – ad es., non è separata dal codice
    - è anche possibile mischiare le due modalità – il contenitore applica prima la configurazione basata su Java e poi quella XML (che potrebbe sovrascrivere gli effetti della prima)
    - in ogni caso, lo sviluppatore deve specificare in dettaglio tutte le informazioni di configurazione

- Una diversa configurazione

```
/* Un'altra configurazione Spring per l'applicazione Show. */
@Bean
public Artist may() {
    return new Musician("Brian", redspecial());
}
@Bean
public Instrument redspecial() {
    Guitar redspecial = new Guitar();
    redspecial.setSound("La la laa");
    return redspecial;
}
```

- le due configurazioni possono convivere, anche nella stessa classe **ShowConfig**

- Componenti e autowiring

- Con Spring, la specifica dei metadati di configurazione – ovvero la specifica dei bean e delle loro dipendenze e relazioni – può avvenire anche sulla base di un'ispezione statica del codice dell'applicazione e di meccanismi di configurazione automatici (impliciti)
  - la definizione dei componenti e la scansione automatica dei componenti semplificano la specifica dei bean
  - l'autowiring semplifica la specifica dei collegamenti tra i bean
  - scansione dei componenti e autowiring sono due meccanismi separati, ma che vengono spesso utilizzati insieme
  - il vantaggio principale di questi meccanismi è la semplificazione della specifica dei metadati di configurazione
  - ci sono però alcune limitazioni – ad es., può essere difficile specificare un insieme di bean e come collegarli – ma in questi casi si possono usare i meccanismi espliciti descritti in precedenza

Componenti

- Un *componente* (nel senso di Spring) è una classe annotata con l'annotazione **@Component**
  - in Spring, un "componente" è intuitivamente un bean che può essere identificato e configurato automaticamente
  - oltre a **@Component**, Spring definisce altre annotazioni (che estendono **@Component**), per tipi specifici di componenti
    - ad es., **@Controller**, **@Repository** e **@Service**
  - inoltre, l'annotazione **@ComponentScan** nella classe di configurazione Java abilita la scansione automatica dei componenti (che vengono a quel punto considerati dei bean)
  - nel seguito, un po' impropriamente, consideriamo componente e bean come sinonimi

Autowiring

- Il collegamento automatico tra componenti (*autowiring*) riguarda l'iniezione automatica delle dipendenze in un componente
  - si basa sull'annotazione **@Autowired**, usata nella dichiarazione di un campo (una variabile d'istanza, anche privata) che specifica una dipendenza di un componente, indica che il contenitore deve identificare automaticamente e assegnare un valore al campo, per soddisfare la dipendenza
    - questo avviene sulla base del tipo o del nome del campo
    - se il tipo di un campo è un'interfaccia e ci sono più componenti che implementano quell'interfaccia, ci potrebbe essere un'ambiguità – che si può risolvere usando l'annotazione **@Primary** sul componente di interesse, oppure usando i qualificatori (che però vanno oltre gli scopi di questa introduzione)
  - **@Autowired** si può usare anche in corrispondenza degli argomenti di un costruttore o di un metodo set

Autowiring

- Con l'autowiring, è spesso utile poter specificare anche dei valori da usare nell'inizializzazione dei componenti – questo può essere fatto mediante un file di configurazione e l'annotazione **@Value**
  - l'annotazione **@Value** può essere usata per specificare il valore (letterale o costante) da assegnare a un campo (una variabile d'istanza, anche privata) di un componente oppure da usare per l'argomento di un costruttore o di un metodo set
  - una forma comune è **@Value("${property.name}")** in cui **property.name** è il nome di una proprietà specificata in un file di proprietà (è un file di configurazione testuale)
    - in questo caso, nella classe di configurazione Java va usata anche l'annotazione **@PropertySource** per specificare qual è il file di configurazione delle proprietà

Esempio

- Riprendiamo il nostro esempio per mostrare l'uso dei componenti e dell'autowiring
  - la definizione delle interfacce **Artist** e **Instrument** è come prima

```
package asw.spring.show;

/* Un artista. */
public interface Artist {

    /* Esibizione dell'artista. */
    public String perform();

}
```

```
package asw.spring.show;

/* Uno strumento musicale. */
public interface Instrument {

    /* Suona lo strumento. */
    public String play();

}
```


Il componente Guitar

- L'implementazione **Guitar** di **Instrument** viene annotata con **@Component** e richiede l'uso di **@Value**

```
package asw.spring.show;

import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Value;

@Component(value="stratocaster")
public class Guitar implements Instrument {

    @Value("${show.stratocaster.sound}")
    private String sound;

    public String play() {  return sound;  }

}
```

Il componente Musician

- L'implementazione **Musician** richiede anche l'uso di **@Autowired**

```
package asw.spring.show;

import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;

@Component(value="hendrix")
public class Musician implements Artist {

    @Value("${show.hendrix.name}")
    private String name;

    @Autowired
    @Qualifier("stratocaster")
    private Instrument instrument;

    public String perform() { "I'm " + name + ": " + instrument.play(); }

}
```


Classe di configurazione

- La nuova classe di configurazione **ShowConfig** (ora le informazioni importanti sono nelle annotazioni)

```
package asw.spring.show.config;

import asw.spring.show.*;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.PropertySource;

/* Configurazione Spring per l'applicazione Show. */
@Configuration
@ComponentScan("asw.spring.show")
@PropertySource("classpath:config.properties")
public class ShowConfig {

}
```

- Il file config.properties

```
show.hendrix.name=Jimi
show.stratocaster.sound=Ua ua uaa
```

La classe Main

- La classe **Main** è come prima

```
package asw.spring.show;

import asw.spring.show.config.ShowConfig;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.
           AnnotationConfigApplicationContext;

/* Applicazione che ottiene e avvia il client. */
public class Main {

   /* Ottiene e avvia un oggetto Client. */
   public static void main(String[] args) {
      ApplicationContext context =
          new AnnotationConfigApplicationContext(ShowConfig.class);
      Artist hendrix = (Artist) context.getBean("hendrix");
      System.out.println( hendrix.perform() );
   }

}
```


Discussione

- Nella configurazione basata su componenti e autowiring
  - le classi per i componenti non sono più strettamente dei POJO
    - i componenti richiedono l'uso di annotazioni Spring per i metadati di configurazione – anche se queste classi non devono estendere né implementare nessun tipo predefinito delle API di Spring
  - usando solo componenti e autowiring c'è una corrispondenza diretta tra classi Java, componenti e bean (questo vincolo non sussiste nelle configurazioni esplicite)
    - nell'esempio, non è possibile avere più bean dalle stesse classi (come due musicisti che suonano strumenti diversi)
  - il client di un bean può ancora accedere al bean mediante l'application context, sulla base del nome (oppure del tipo) del bean di interesse

* Gestione delle dipendenze

- Un altro aspetto importante delle applicazioni complesse sono le *dipendenze tra moduli*
  - attenzione, si tratta di una nozione diversa dalle *dipendenze tra bean e componenti* di cui abbiamo parlato finora
- In molti casi, un progetto software utilizza delle funzionalità riusabili (sotto forma di librerie) ed è suddiviso in più parti, per comporre un progetto modulare
  - ogni parte di un progetto o libreria costituisce un *modulo*
  - ogni modulo può avere delle *dipendenze* da altri moduli
  - la *gestione delle dipendenze* è una tecnica per dichiarare, risolvere e usare le dipendenze richieste da un progetto software, in modo automatizzato
  - le dipendenze di un progetto software vengono di solito gestite tramite strumenti di build automation – come *Gradle* oppure *Maven*

Gestione delle dipendenze

- In particolare, le applicazioni basate sul framework Spring dipendono dalla presenza di alcune librerie (file jar, in versioni specifiche) nel classpath dell'applicazione – per la compilazione, l'esecuzione e/o i test
  - ciascuna di queste librerie o risorse è una *dipendenza* dell'applicazione – da non confondere con le dipendenze di bean e componenti
    - ad es., le applicazioni Spring mostrate finora dipendono dalla libreria org.springframework:spring-context (che si occupa dell'iniezione delle dipendenze) – e anche da org.springframework:spring-test per i test (che per semplicità non sono stati mostrati)
    - una semplice applicazione web Spring richiede spesso una dozzina di dipendenze o più

Gestione delle dipendenze con Gradle

- Con Gradle, le dipendenze vengono specificate in un file di nome `build.gradle`, nel blocco `dependencies` – ad esempio

```
apply plugin: 'java'
java {
    sourceCompatibility = '17'
}

repositories {
    mavenCentral()
}

apply plugin: 'application'
application {
    mainClass = 'asw.spring.show.Main'
}

dependencies {
    implementation 'org.springframework:spring-context:6.0.12'
    testImplementation 'org.springframework:spring-test:6.0.12'
    testImplementation 'org.junit.jupiter:junit-jupiter:5.10.0'
    testRuntimeOnly('org.junit.platform:junit-platform-launcher')
}

test {
    useJUnitPlatform()
}
```


* Discussione

- Spring è un framework applicativo modulare, per lo sviluppo di applicazioni Java di tipo enterprise
  - questa dispensa ha descritto l'iniezione delle dipendenze, che è una caratteristica fondamentale di Spring, implementata dai moduli fondamentali ("core") del framework
  - Spring è un framework composto da una ventina di moduli – i moduli di Spring riguardano, tra l'altro
    - lo sviluppo di applicazioni web (Spring Web MVC)
    - l'accesso ai dati (per esempio con JDBC o mediante ORM) – Spring Data
    - il messaging e l'integrazione (Spring Integration)
    - ulteriore supporto e semplificazione nello sviluppo di applicazioni basate su Spring – Spring Boot
    - supporto per lo sviluppo di applicazioni distribuite per il cloud – Spring Cloud