

Luca Cabibbo
Architettura
dei Sistemi
Software

Spring Boot

dispensa asw825
marzo 2021

*Previous to Spring Boot,
I remember XML hell, confusing set up,
and many hours of frustration.*

Greg Turnquist



- Riferimenti

- ❑ Spring Boot
 - <https://spring.io/projects/spring-boot>
- ❑ **Spring Boot Reference Guide, 2.4.4, 2021.**
 - <https://docs.spring.io/spring-boot/docs/current/reference/html/>
 - <https://docs.spring.io/spring-boot/docs/2.4.4/reference/html/>
- ❑ Walls, C. **Spring Boot in Action**. Manning, 2016.
- ❑ Walls, C. **Spring in Action**, fifth edition. Manning, 2019.



- Obiettivi e argomenti

□ Obiettivi

- fornire un'introduzione a Spring Boot
- presentare alcuni ulteriori progetti Spring nel contesto di Spring Boot

□ Argomenti

- introduzione a Spring Boot
- applicazioni web con Spring Web MVC
- Spring Data JPA
- Spring Boot Actuator
- configurazione tramite proprietà e profili
- discussione



* Introduzione a Spring Boot

- **Spring Boot** è un progetto Spring che ha lo scopo di rendere più semplice lo sviluppo e l'esecuzione di applicazioni Spring
 - un'applicazione Spring può richiedere una gran quantità di metadati di configurazione – anche se si utilizzano componenti e autowiring
 - Spring Boot semplifica lo sviluppo delle applicazioni, perché ne effettua una configurazione automatica (ove possibile), sulla base di valori di default “intelligenti” – un'applicazione Spring Boot richiede, di solito, solo una configurazione minima
 - a tal fine, Spring Boot utilizza un approccio “opinionated” (“convenzionale”, basato su opinioni e convenzioni proprie)
 - le scelte di default possono essere comunque sovrascritte mediante delle configurazioni esplicite
 - Spring Boot fornisce inoltre delle opzioni per la costruzione (build) e il rilascio (deploy) delle applicazioni in produzione



Elementi essenziali di Spring Boot

- Alcune caratteristiche principali di Spring Boot – sono relative ad aspetti diversi
 - starter dependencies – configurazione automatica delle librerie e delle dipendenze dell'applicazione (da non confondere con l'iniezione delle dipendenze)
 - configurazione automatica di bean e componenti, e delle relazioni tra di essi (anche di bean non dichiarati esplicitamente) – anche sulla base delle dipendenze (librerie) utilizzate
 - actuator – per ispezionare un'applicazione Spring Boot in esecuzione



- Un primo semplice esempio

- Lo sviluppo di un'applicazione Spring Boot inizia con la creazione di un progetto, della struttura delle sue cartelle, nonché un file per la costruzione (build) dell'applicazione (ad es., Gradle o Maven)
 - questa attività può essere svolta con *Spring Initializr* – <https://start.spring.io/> – a partire dalla scelta del tipo di applicazione (e delle corrispondenti dipendenze starter) – ad es., un'applicazione web
 - l'Initializr genera il progetto, che può essere scaricato sul proprio computer
 - il progetto contiene la struttura delle cartelle, la classe per l'applicazione, una classe di test, un file per le proprietà dell'applicazione (vuoto) e il file di build Gradle o Maven
 - l'Initializr può anche essere acceduto da un plugin dell'IDE
 - a questo punto si è pronti per iniziare lo sviluppo dell'applicazione



File e cartelle generati da Spring Initializr

- Ecco la struttura di file e cartelle generati dall'Initializr per un'applicazione minimale

```
|
+--- build.gradle (oppure pom.xml)
+--- settings.gradle (con il nome del progetto)
\--- src
    +--- main
        |   +--- java
        |   |   \--- asw.springboot.hello
        |   |       \--- HelloApplication.java
        |   \--- resources
        |       \--- application.properties
    \--- test
        \--- java
            \--- asw.springboot.hello
                \--- HelloApplicationTests.java
```



File di build

- Questo è un estratto del file di build `build.gradle` per Gradle

```
plugins {
    id 'org.springframework.boot' version '2.4.4'
    id 'io.spring.dependency-management' version '1.0.10.RELEASE'
    id 'java'
}

group = 'asw.springboot'
sourceCompatibility = '11'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
    useJUnitPlatform()
}
```



Gestione delle dipendenze

- In generale, le applicazioni 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
 - ad es., le applicazioni Spring dipendono in genere dalle librerie **spring-core** e **spring-context** (quest'ultima si occupa dell'iniezione delle dipendenze)
 - una dipendenza può essere **diretta** (l'applicazione dipende da X) oppure **indiretta** o **transitiva** (se l'applicazione dipende da X e X dipende da Y, allora l'applicazione dipende anche da Y)
 - le dipendenze transitive sono le più difficili da identificare e da gestire



Dipendenze starter

- Spring Boot semplifica la gestione delle dipendenze fornendo e supportando ufficialmente un insieme curato di dipendenze starter
 - una **dipendenza starter** (**starter dependency**) è una dipendenza (di solito ampia e transitiva) – la cui inclusione implica automaticamente l'inclusione delle sue dipendenze transitive
 - grazie a esse, un'applicazione Spring Boot richiede di solito poche dipendenze – nell'esempio, **spring-boot-starter** e **spring-boot-starter-test** – mentre un'applicazione Spring tradizionale richiede spesso una dozzina di dipendenze o più
 - ad es., **spring-boot-starter** implica **spring-boot**, che implica **spring-core** e **spring-context** (che si occupa dell'iniezione delle dipendenze), nonché alcune dipendenze per il logging
 - inoltre, **spring-boot-starter-test** implica (transitivamente) alcune dipendenze fondamentali comunemente utilizzate per i test – come **junit** (per i test unitari), **mockito-core** (per i test di integrazione) e **hamcrest-core** (per le asserzioni)



Un'applicazione Spring Boot

- Questa è la classe principale per l'applicazione (predefinita)

```
package asw.springboot.hello;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloApplication {

    public static void main(String[] args) {
        SpringApplication.run( HelloApplication.class, args );
    }

}
```



Esecuzione di un'applicazione Spring Boot

- Come viene eseguita un'applicazione Spring Boot?
 - la classe **SpringApplication** (di Spring Boot) viene utilizzata per effettuare il “bootstrap” e l'avvio dell'applicazione – tra l'altro, crea un application context per l'applicazione
 - l'annotazione **@SpringBootApplication** usata per la classe principale dell'applicazione indica una combinazione di
 - **@Configuration** – che etichetta la classe come una classe di configurazione Java
 - **@ComponentScan** – che abilita la scansione e l'identificazione automatica dei componenti/bean
 - **@EnableAutoConfiguration** – si occupa della creazione automatica dei componenti/bean mancanti o necessari (secondo Spring Boot), sulla base dell'application context e delle dipendenze specificate per l'applicazione



Esecuzione dell'applicazione Hello

- L'applicazione può essere eseguita, così come è
 - l'esecuzione può essere avviata con il comando `gradle bootRun` – non con il comando `gradle run`
 - in alternativa è possibile costruire il jar dell'applicazione (`gradle build`) e poi eseguirla con `java -jar hello.jar`
 - in questo caso, l'applicazione genera semplicemente un log e termina

```

  ____  _
 / ___|| | | |
 \___ \| |_| |
  ___) | | | |
 |___) | |_| |
  _____|_|_|_|_|
:: Spring Boot ::      (v2.2.6.RELEASE)

```

```

2020-03-27 15:09:30.167 INFO 26646 --- [ main] asw.springboot.hello.HelloApplication : Starting HelloApplication on workstation with PID 26646
2020-03-27 15:09:30.168 INFO 26646 --- [ main] asw.springboot.hello.HelloApplication : No active profile set, falling back to default profiles: default
2020-03-27 15:09:30.580 INFO 26646 --- [ main] asw.springboot.hello.HelloApplication : Started HelloApplication in 0.726 seconds (JVM running for 1.024)

```



Comportamento specifico

- Per specificare del codice da eseguire con l'applicazione, è possibile definire un componente/bean come il seguente

```

package asw.springboot.hello;

import org.springframework.stereotype.Component;
import org.springframework.boot.CommandLineRunner;

@Component
public class HelloRunner implements CommandLineRunner {

    public void run(String[] args) {
        System.out.println("Hello, world!");
    }

}

```

```

  ____  _
 / ___|| | | |
 \___ \| |_| |
  ___) | | | |
 |___) | |_| |
  _____|_|_|_|_|
:: Spring Boot ::      (v2.2.6.RELEASE)

```

```

2020-03-27 15:11:40.315 INFO 26729 --- [ main] asw.springboot.hello.HelloApplication : Starting HelloApplication on workstation with PID 26729
2020-03-27 15:11:40.317 INFO 26729 --- [ main] asw.springboot.hello.HelloApplication : No active profile set, falling back to default profiles: default
2020-03-27 15:11:40.734 INFO 26729 --- [ main] asw.springboot.hello.HelloApplication : Started HelloApplication in 0.72 seconds (JVM running for 1.018)
2020-03-27 15:11:40.735 INFO 26729 --- [ main] class asw.springboot.hello.HelloRunner : Hello, world!

```



Scopo di un'applicazione Spring Boot

- Intuitivamente, lo scopo di un'applicazione Spring Boot è fare in modo che vengano creati e avviati/attivati tutti i componenti/bean dell'applicazione
 - questi componenti vanno in genere definiti separatamente dalla classe principale dell'applicazione (quella annotata **@SpringBootApplication**)



* Applicazioni web con Spring Web MVC

- Consideriamo ora lo sviluppo di un'applicazione web con Spring Boot – e il framework Spring Web MVC – iniziando da un'applicazione molto semplice
 - con Spring Initializr va selezionata la dipendenza “Web”
 - in questo modo viene utilizzata la dipendenza starter **spring-boot-starter-web** (anziché la più generica **spring-boot-starter**)

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
}
```




La dipendenza starter web

- Ecco alcune implicazioni della dipendenza `spring-boot-starter-web`
 - la dipendenza starter `spring-boot-starter-web` implica `spring-boot-starter` (già discussa prima), che come sappiamo implica `spring-core` e `spring-context` (per l'iniezione delle dipendenze)
 - la dipendenza `spring-boot-starter-web` implica anche `spring-webmvc` (il framework Spring Web MVC, discusso più avanti)
 - inoltre, `spring-boot-starter-web` implica `tomcat-embed-core` – pertanto l'applicazione potrà essere eseguita in un application server Tomcat embedded
 - questa è una scelta convenzionale (“opinionated”) di Spring Boot – che volendo può essere sovrascritta e modificata (ma in questo caso, per semplicità, ci va bene così)



Struttura di un'applicazione web

- In un'applicazione web vengono create, tra le risorse, anche due cartelle (inizialmente vuote) per i contenuti statici (`static`) e dinamici (`templates`) dell'applicazione

```
|
+--- build.gradle (oppure pom.xml)
+--- settings.gradle
\--- src
    +--- main
        |   +--- java
        |   |   \--- asw.springboot.web.hello
        |   |       \--- HelloApplication.java
        |   \--- resources
        |       +--- static
        |       +--- templates
        |       \--- application.properties
    \--- test
        \--- java
            \--- asw.springboot.web.hello
                \--- HelloApplicationTests.java
```



Classe principale per l'applicazione

- La classe principale per l'applicazione è ancora come prima

```
package asw.springboot.web.hello;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloApplication {

    public static void main(String[] args) {
        SpringApplication.run( HelloApplication.class, args );
    }

}
```

- infatti lo scopo di questa applicazione è semplicemente fare in modo che vengano creati e avviati/attivati tutti i componenti/bean dell'applicazione



Proprietà dell'applicazione

- Le applicazioni Spring Boot possono essere configurate mediante dei file di proprietà – in cui è possibile specificare sia le proprietà comuni di Spring Boot (ad es., `server.port`) che delle proprietà specifiche dell'applicazione
 - le proprietà dell'applicazione vanno specificate nel file `application.properties` (oppure nel file `application.yml`, con una sintassi diversa)
 - esempio di file `application.properties`

```
# application.properties
server.port=8080
```

- in effetti, Spring Boot definisce dei valori di default (spesso sensati) per le proprietà comuni – e dunque non sempre è necessario configurare tutte le proprietà dell'applicazione
 - ad es., il valore di default della proprietà `server.port` è proprio `8080` – ma, volendo, può essere modificato



Esecuzione dell'applicazione web Hello

- L'applicazione web può essere eseguita così come è, anche senza nessun componente/bean aggiuntivo
 - in questo caso, dal log generato dall'applicazione si può evincere che
 - viene avviato Tomcat, sulla porta 8080
 - al suo interno viene eseguita la nostra semplice applicazione web
 - sono stati definiti anche degli handler associati a degli URL e path
 - l'applicazione è pronta ad accettare richieste
 - per ora, essendo l'applicazione vuota, l'accesso a <http://localhost:8080/> porta a una pagina di errore generata da Tomcat – questo indica che Tomcat è effettivamente in ascolto
 - in alternativa, l'applicazione può essere assemblata come un WAR e rilasciata in un application server separato



Classe di test per l'applicazione

- Ecco la classe di test definita da Spring Initializr
 - è solo una classe scheletro di test di esempio
 - il test (anche se vuoto) verifica se il caricamento del contesto dell'applicazione avviene senza problemi – ad es., che tutti i bean possono essere inizializzati

```
package asw.springboot.web.hello;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
public class HelloApplicationTests {

    @Test
    public void contextLoads() {
    }

}
```



- Personalizzazione dell'applicazione Hello

- Ecco un semplice componente che è un controller web (in virtù di **@Controller**) per accettare richieste al path **/hello**

```
package asw.springboot.web.hello;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HelloController {

    @RequestMapping("/hello")
    public @ResponseBody String hello() {
        return "Hello, world!";
    }
}
```

- una richiesta GET **http://localhost:8080/hello** restituisce la stringa **Hello, world!**



Personalizzazione dell'applicazione Hello

- Alcune indicazioni per comprendere questa classe
 - l'annotazione **@Controller** indica un tipo di **@Component** – un controller Spring Web MVC, per ricevere richieste web
 - l'annotazione **@RequestMapping** associa un metodo del controller web a un'operazione HTTP (GET, di default) per il path specificato (nell'esempio, **/hello**)
 - l'annotazione **@ResponseBody** specifica che il valore restituito dal metodo va interpretato come il contenuto della risposta
 - altrimenti, in Spring Web MVC, il valore restituito da un metodo di un controller viene interpretato come il nome della vista da visualizzare al termine dell'esecuzione del metodo (discusso più avanti)
 - per questo, una richiesta GET **http://localhost:8080/hello** restituisce la stringa **Hello, world!**



- Il framework Spring Web MVC

- Il framework *Spring Web MVC* definisce la struttura e il modello di programmazione delle applicazioni web con Spring – basato sul pattern MVC (Model-View-Controller) e composto da
 - oggetti *controller* – responsabili di elaborare le richieste web degli utenti
 - il *modello* – responsabile di gestire le informazioni di interesse dell'applicazione
 - *viste* – responsabili di visualizzare le risposte e le informazioni del modello agli utenti



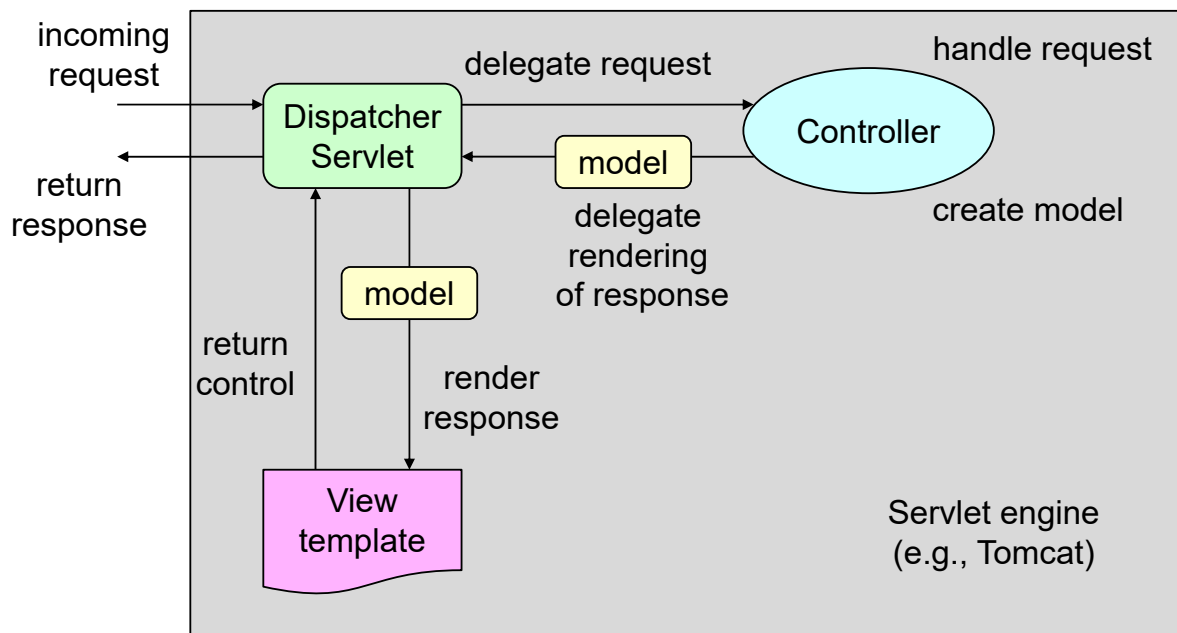
Il framework Spring Web MVC

- Nel framework Spring Web MVC
 - il *modello* è una semplice mappa `Map<String, Object>` da attributi a valori – oppure si può usare un oggetto `Model` con operazioni `addAttribute` e `getAttribute`
 - ciascun *controller* svolge di solito queste attività
 - riceve una richiesta con i suoi parametri
 - elabora la richiesta e popola il modello
 - restituisce il nome della vista da utilizzare per il rendering della risposta
 - ciascuna *vista* è un template di pagine web
 - le viste possono essere implementate con diverse tecnologie (come JSP e Thymeleaf)
 - il rendering di una vista è di solito basato sulla sostituzione di elementi del template con i valori degli attributi del modello



Gestione delle richieste

- Ecco una descrizione di alto livello della modalità di gestione di una richiesta



27

Spring Boot

Luca Cabibbo ASW



- Un altro esempio

- Mostriamo ora un altro piccolo esempio, basato su Spring Web MVC, con viste Thymeleaf
 - in questo caso va utilizzata anche la dipendenza `spring-boot-starter-thymeleaf` – si consulti la documentazione di Spring Boot per un elenco delle possibili dipendenze starter

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

- vogliamo realizzare un'applicazione per gestire richieste di tipo `/hello/name`, che restituisce un saluto personalizzato
 - usiamo un controller per `/hello/name`, un modello con solo un attributo `name` e una vista `greeting`

28

Spring Boot

Luca Cabibbo ASW



Controller per l'applicazione Hello

- Ecco il controller per gestire richieste di tipo `/hello/name`

```
package asw.springboot.web.hello;

import java.util.Map;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.PathVariable;

@Controller
public class HelloController {

    @RequestMapping("/hello/{name}")
    public String hello(Map<String, Object> model,
                       @PathVariable String name) {
        model.put("name", name);
        return "greeting";
    }
}
```

- il controller riceve il nome da salutare come parametro (`@PathVariable`), lo copia nell'attributo `name` del modello e delega la risposta alla vista `greeting`



Vista per l'applicazione Hello

- Ecco la vista Thymeleaf `greeting` – specificata dal file `greeting.html` nella cartella `templates`

```
<html>
<body>

    <p>Hello, <span th:text="${name}">name goes here</span>!</p>

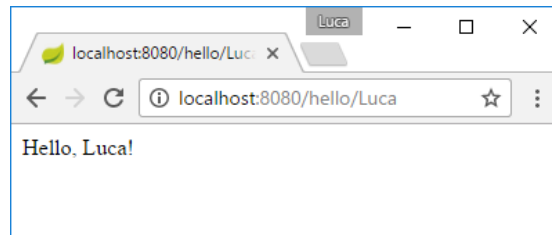
</body>
</html>
```

- questa vista ha un elemento `` associato all'attributo `name` del modello
- il rendering di questa vista è basato sulla sostituzione del contenuto fittizio di questo elemento (nell'esempio, "`name goes here`") con il contenuto dell'attributo `name` nel modello (in virtù dell'attributo `th:text="${name}"` dell'elemento `span`)



L'applicazione Hello

- Una richiesta GET <http://localhost:8080/hello/Luca>



```
<html>
<body>
  <p>Hello, <span>Luca</span>!</p>
</body>
</html>
```



Configurazione automatica

- Di solito, un'applicazione web basata sul framework Spring Web MVC richiede una configurazione complessa
 - in termini di configurazione Java – ad es., una classe per inizializzare l'applicazione web, che crea la dispatcher servlet
 - oppure di configurazione XML – ad es., un file `web.xml`
- Viceversa, con Spring Boot, la configurazione di un'applicazione web è molto più semplice – nell'esempio, non è necessario nessun altro file oltre a quelli già mostrati
 - è poi Spring Boot a occuparsi di identificare, creare e configurare (automaticamente e in modo intelligente) tutti i componenti/bean necessari per l'applicazione
 - in particolare, crea automaticamente tutti i bean MVC (`DispatcherServlet`, `HandlerMapping`, `Adapter`, `ViewResolver`) – in questo caso il `ViewResolver` è un `ThymeleafViewResolver` (configurato automaticamente, in virtù della dipendenza starter)



Scope di modello e controller



- Osservazioni sullo scope degli elementi in Spring Web MVC
 - il modello ha scope *request* – la dispatcher servlet prepara, per ogni richiesta, un nuovo modello per il controller selezionato
 - dunque, in genere, gli attributi del modello non sono condivisi tra richieste diverse – tuttavia, è possibile specificare con **@SessionAttributes** che alcuni attributi del modello abbiano scope *session* – per gestire lo stato delle sessioni come attributi del modello
 - il controller ha scope *singleton* (per default)
 - dunque c'è una condivisione delle variabili d'istanza del controller tra tutte le richieste e tutte le sessioni gestite *da quel controller* – ma non tra controller diversi
 - tuttavia, è possibile specificare per il controller lo scope *session* – per gestire lo stato delle sessioni come variabili d'istanza del controller (se tutte le operazioni del caso d'uso sono implementate da un solo controller)

33

Spring Boot

Luca Cabibbo ASW



- Test di un'applicazione web

- Il test di un'applicazione Spring Web MVC può essere basato sulla verifica di diversi aspetti
 - ad es., con riferimento a una richiesta GET `/hello/Luca`
 - che ci sia una risposta HTTP OK
 - che il nome della vista restituita dal controller sia *greeting*
 - che il modello restituito dal controller contenga un attributo *name*
 - che il valore dell'attributo *name* nel modello sia *Luca*
 - che il contenuto della pagina HTTP restituita contenga la stringa `Hello, Luca!`

34

Spring Boot

Luca Cabibbo ASW



Un esempio di test

```
package asw.springboot.web.hello;

import ...;

@WebMvcTest(HelloController.class)
public class HelloApplicationMockMvcWebTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void helloLucaTest() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.get("/hello/Luca"))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.view().name("greeting"))
            .andExpect(MockMvcResultMatchers.model().attributeExists("name"))
            .andExpect(MockMvcResultMatchers.model().attribute("name", "Luca"))
            .andExpect(MockMvcResultMatchers.content()
                .string(containsString("Hello, <span>Luca</span>!")));
    }
}
```

35

Spring Boot

Luca Cabibbo ASW



* Spring Data JPA

- **Spring Data** è un progetto Spring (composto a sua volta da altri progetti) di supporto alla gestione di oggetti persistenti e all'accesso alle basi di dati
 - in particolare, il progetto **Spring Data JPA** supporta l'implementazione di repository basati su JPA
 - in JPA, un' **entità** è un tipo di oggetti persistenti – gli oggetti persistenti vanno etichettati con l'annotazione **@Entity**
 - un **repository** è un oggetto che fornisce un'interfaccia CRUD per l'accesso a un'entità nella base di dati
 - Spring Data fornisce i **repository dinamici**
 - lo sviluppatore ne definisce solo l'interfaccia – e l'implementazione viene realizzata automaticamente da Spring Data
 - diversi sottoprogetti di Spring Data realizzano implementazioni differenti, ad es., per JPA o per MongoDB

36

Spring Boot

Luca Cabibbo ASW



Utilizzo di Spring Data JPA

- Per utilizzare Spring Data JPA va utilizzata la dipendenza starter **spring-boot-starter-data-jpa**
 - questa dipendenza implica transitivamente l'uso di Hibernate come provider JPA
 - anche questa è una scelta convenzionale (“opinionated”) di Spring Boot, che può essere sovrascritta e modificata
 - va invece aggiunta separatamente una dipendenza per il driver per il database
 - ad es., la dipendenza **org.hsqldb:hsqldb** per usare HSQL (un in-memory db, utile durante lo sviluppo e i test)

```
dependencies {  
    ...  
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
    implementation 'org.hsqldb:hsqldb'  
    ...  
}
```



Esempio: RestaurantServiceApplication

- Si vuole definire (nell'ambito di un'applicazione **efood** per la gestione di un servizio di ordinazione e spedizione a domicilio di pasti da ristoranti, su scala nazionale) un servizio applicativo **restaurant-service** per la gestione di un insieme di ristoranti
 - i ristoranti sono definiti come un'entità JPA **Restaurant**
 - l'accesso ai ristoranti avviene mediante un repository **RestaurantRepository**
 - viene inoltre definito un servizio **RestaurantService**
 - l'applicazione viene realizzata (per ora) come un'applicazione Spring Web MVC
 - questo esempio verrà infatti ripreso anche in successive dispense, per esemplificare altre modalità di interazione con l'applicazione e altre tecnologie



Architettura esagonale del servizio

- Il servizio applicativo **restaurant-service** di **efood** viene strutturato con l'architettura esagonale, usando i seguenti package
 - **asw.efood.restaurantservice** è il package di base del servizio
 - **asw.efood.restaurantservice.domain** definisce l'interno (logica di business) del servizio, comprese le sue porte
 - contiene le entità, i service e i repository (le interfacce) – service e repository sono porte
 - package per gli adattatori
 - **asw.efood.restaurantservice.web** definisce l'adattatore web utilizzato dagli utenti per accedere al servizio – contiene i controller web e i presentation model
 - non è necessario definire l'adattatore per JPA, in virtù dell'uso dei repository dinamici
 - in seguito verranno definiti altri adattatori per questo servizio, ciascuno in un proprio package



L'entità Restaurant

```
package asw.efood.restaurantservice.domain;

import javax.persistence.*;

@Entity
public class Restaurant {

    @Id
    @GeneratedValue
    private Long id;

    private String name;
    private String location;

    ... costruttori e metodi get, set e toString ...

}
```



Repository per Restaurant

- In Spring, un repository per un'entità viene definito come un'interfaccia che estende **CrudRepository<Entity,Id>**
 - questa interfaccia definisce metodi come **save**, **delete** e **findById** – a cui è possibile aggiungere altri metodi (usando uno schema convenzionale per i nomi)

```
package asw.efood.restaurantservice.domain;

import org.springframework.data.repositoryCrudRepository;
import java.util.*;

public interface RestaurantRepository
    extends CrudRepository<Restaurant, Long> {

    public Restaurant findByName(String name);
    public Collection<Restaurant> findAll();
    public Collection<Restaurant> findAllByLocation(String location);
}
```

- l'implementazione di questa interfaccia è fornita da Spring Data come un repository dinamico

41

Spring Boot

Luca Cabibbo ASW



Un servizio per gestire i ristoranti

- In genere è utile definire una classe “servizio” per esporre le funzionalità dell'applicazione – ad es., nei confronti dei controller
 - i servizi sono una tipologia di componenti, annotati **@Service**

```
package asw.efood.restaurantservice.domain;

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.beans.factory.annotation.Autowired;
import java.util.*;

@Service
@Transactional
public class RestaurantService {

    @Autowired
    private RestaurantRepository restaurantRepository;

    ... operazioni del servizio ...
}
```

42

Spring Boot

Luca Cabibbo ASW



Un servizio per gestire i ristoranti

- Alcune operazioni del servizio per la gestione dei ristoranti

```
public Restaurant createRestaurant(String name, String location) {
    Restaurant restaurant = new Restaurant(name, location);
    restaurant = restaurantRepository.save(restaurant);
    return restaurant;
}

public Restaurant getRestaurant(Long id) {
    Restaurant restaurant =
        restaurantRepository.findById(id).orElse(null);
    return restaurant;
}

public Collection<Restaurant> getAllRestaurants() {
    Collection<Restaurant> restaurants =
        restaurantRepository.findAll();
    return restaurants;
}
```



Alcune operazioni web

- Esaminiamo ora la definizione di alcune semplici operazioni del controller

```
package asw.efood.restaurantservice.web;

import asw.efood.restaurantservice.domain.*;
import org.springframework.stereotype.Controller;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import org.springframework.ui.Model;
import java.util.*;

@Controller
@RequestMapping(path="/web")
public class RestaurantWebController {

    @Autowired
    private RestaurantService restaurantService;

    ...
}
```



Alcune operazioni web

□ Ricerca di un ristorante

```
/* Trova il ristorante con restaurantId. */
@GetMapping("/restaurants/{restaurantId}")
public String getRestaurant(Model model,
                           @PathVariable Long restaurantId) {
    Restaurant restaurant =
        restaurantService.getRestaurant(restaurantId);
    model.addAttribute("restaurant", restaurant);
    return "get-restaurant";
}
```

- le annotazioni **@GetMapping** e **@PostMapping** possono essere usate al posto di **@RequestMapping(path="...", method=RequestMethod.GET)** e di **@RequestMapping(path="...", method=RequestMethod.POST)**



Alcune operazioni web

□ Ecco la vista Thymeleaf `get-restaurant`

```
<html>
<body>

<h1><span th:text="${restaurant.name}">name goes here</span></h1>

<p>
    Restaurant <span th:text="${restaurant.name}">name</span>
    (id=<span th:text="${restaurant.id}">id</span>)
    is located in <span th:text="${restaurant.location}">location</span>.
</p>

</body>
</html>
```

- si noti l'uso della notazione *object.field* per accedere a un campo di un oggetto passato come attributo del modello
 - in pratica, queste espressioni vengono valutate invocando i metodi *getField()* dell'oggetto



Alcune operazioni web

□ Elenco di tutti i ristoranti

```
/* Trova tutti i ristoranti. */
@GetMapping("/restaurants")
public String getRestaurants(Model model) {
    List<Restaurant> restaurants =
        restaurantService.getAllRestaurants();
    model.addAttribute("restaurants", restaurants);
    return "get-restaurants";
}
```



Alcune operazioni web

□ Ecco (parte del) la vista Thymeleaf `get-restaurants`

```
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Location</th>
      <th>Id</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="restaurant : ${restaurants}">
      <td th:text="${restaurant.name}">name</td>
      <td th:text="${restaurant.location}">location</td>
      <td><a th:href="@{'/web/restaurants/'+${restaurant.id}}">
        <span th:text="${restaurant.id}">id</span></a>
      </td>
    </tr>
  </tbody>
</table>
```

- si noti l'iterazione `th:each="restaurant : ${restaurants}"`



Alcune operazioni web

□ Aggiunta di un nuovo ristorante

- l'aggiunta di un ristorante richiede l'uso di un form per l'inserimento dei dati del ristorante (nome e città)
- il pattern Presentation Model suggerisce di definire una classe che rappresenta i dati del form

```
package asw.efood.restaurantsservice.web;  
  
public class AddRestaurantForm {  
    private String name;  
    private String location;  
  
    ... costruttore e metodi get e set ...  
}
```



Alcune operazioni web

□ Aggiunta di un nuovo ristorante

- nel controller va definita un'operazione per accedere al form per l'inserimento dei dati del ristorante
 - l'operazione deve passare al form un oggetto della classe che rappresenta i dati del form – questo potrebbe anche contenere dati da visualizzare nel form

```
/* Crea un nuovo ristorante (form). */  
@GetMapping(value="/restaurants", params={"add"})  
public String getAddRestaurantForm(Model model) {  
    model.addAttribute("form", new AddRestaurantForm());  
    return "add-restaurant-form";  
}
```

- link alla pagina per aggiungere un nuovo ristorante

```
<a href="/web/restaurants?add">Add a restaurant</a>
```



Alcune operazioni web

□ Il form nella vista Thymeleaf `add-restaurant-form`

```
<form th:action="@{/web/restaurants}" method="POST"
      th:object="${form}">
  <div>
    <label>Name</label>
    <input type="text" th:field="*{name}" required>
  </div>
  <div>
    <label>Location</label>
    <input type="text" th:field="*{location}" required>
  </div>
  <button type="submit">Add restaurant</button>
</form>
```



Alcune operazioni web

□ Aggiunta di un nuovo ristorante

- il controller deve poi definire un'operazione per gestire la ricezione del form con i dati del ristorante

```
/* Crea un nuovo ristorante. */
@PostMapping("/restaurants")
public String addRestaurant(Model model,
    @ModelAttribute("form") AddRestaurantForm form) {
    Restaurant restaurant =
        restaurantService
            .createRestaurant(form.getName(), form.getLocation());
    model.addAttribute("restaurant", restaurant);
    return "get-restaurant";
}
```

- l'operazione riceve come parametro un nuovo oggetto form, che contiene i dati inseriti dall'utente



Configurazione

- Anche in questo caso non è richiesta nessuna ulteriore configurazione oltre a quanto mostrato
 - Spring Data fornisce l'implementazione dinamica dei repository specificati – inoltre, si occupa della gestione delle sorgenti di dati e delle transazioni secondo una configurazione predefinita
 - è anche possibile fornire in modo esplicito tutte le informazioni di configurazione per l'accesso alla base di dati – che sono di solito necessarie per l'accesso a una base di dati “vera”, da usare in produzione



Esercizio

- Estendere il servizio applicativo per la gestione dei ristoranti come segue
 - implementare la gestione dei menu dei ristoranti – il menu **RestaurantMenu** di un ristorante è un elenco di **MenuItem** (ciascuno con un id, un nome e un prezzo)
 - definire un'operazione per trovare il menu di un ristorante (con il relativo template)
 - definire un'operazione per creare o modificare il menu di un ristorante (con il relativo form e la classe per i dati del form)



* Spring Boot Actuator

- *Spring Boot Actuator* fornisce un insieme di caratteristiche aggiuntive per il monitoraggio e la gestione delle applicazioni – da utilizzare anche quando un'applicazione è in produzione
 - fornisce meccanismi di monitoraggio (ad es., tracce e statistiche delle richieste ricevute) e di gestione (ad es., per arrestare remotamente un'applicazione)
 - fornisce inoltre meccanismi di ispezione, per analizzare la configurazione dei bean in esecuzione – ad es., per ispezionare le configurazioni automatiche fatte da Spring Boot
 - tutto sotto forma di risorse e operazioni REST
 - va aggiunta la dipendenza `spring-boot-starter-actuator`

```
dependencies {  
    compile('org.springframework.boot:spring-boot-starter-web')  
    compile('org.springframework.boot:spring-boot-starter-actuator')  
    ...  
}
```



Actuator e endpoint

- Ecco alcuni endpoint predefiniti forniti da Spring Boot Actuator (sono risorse REST)
 - `/actuator` – fornisce un elenco degli endpoint disponibili
 - `/actuator/info` – informazioni sull'applicazione (personalizzabile)
 - `/actuator/health` – stato di salute dell'applicazione (e metriche)
 - `/actuator/beans` – elenco dei bean e delle loro relazioni
 - `/actuator/conditions` – report sulla configurazione automatica dell'applicazione
 - `/actuator/mappings` – elenca i path URI dell'applicazione e i controller a cui sono associati
 - `/actuator/metrics` – metriche sull'utilizzo dell'applicazione e sul consumo di risorse
 - `/actuator/httptrace` – traccia delle ultime (100) richieste HTTP



Actuator e endpoint

- Ulteriori considerazioni sugli endpoint di Spring Boot Actuator
 - gli endpoint possono essere abilitati oppure disabilitati – inoltre possono essere esposti mediante HTTP oppure JMX (oppure anche non esposti)
 - per default, la maggior parte degli endpoint sono abilitati, ma solo pochi sono anche esposti – per la precisione, `/actuator`, `/actuator/info` e `/actuator/health`
 - la configurazione degli endpoint può essere specificata nel file `application.properties` – ad esempio

```
# oltre a info ed health, esponi anche l'endpoint beans  
management.endpoints.web.exposure.include=info,health,beans
```

- inoltre, è possibile personalizzare gli endpoint predefiniti da Spring Boot Actuator, nonché aggiungere dei nuovi endpoint



* Configurazione tramite proprietà e profili

- Molte applicazioni richiedono delle configurazioni complesse
 - qui ci occupiamo della configurazione di un'applicazione basata su proprietà esterne e su profili di configurazione
 - una *proprietà* è un attributo (ovvero, un nome) a cui è associato un valore
 - le proprietà sono utili, ad es., per configurare le credenziali per l'accesso alla base di dati usata dall'applicazione
 - i *profili* di configurazione consentono di definire più configurazioni per una stessa applicazione, da usare in scenari di esecuzione differenti
 - non ci occupiamo, invece, dell'uso delle configurazioni per sovrascrivere esplicitamente le configurazioni automatiche (implicite) di Spring Boot – il che è sempre possibile



Un semplice esempio

- Consideriamo una semplice applicazione web (REST) per accedere a una parola fortunata – questo è il suo controller

```
package asw.springboot.luckyword;

import org.springframework.web.bind.annotation.*;
import org.springframework.beans.factory.annotation.Value;

@RestController
public class LuckyWordController {

    @Value("${lucky.word}")
    private String luckyWord;

    @GetMapping("/lucky-word")
    public String luckyWord() {
        return "The lucky word is: " + luckyWord;
    }
}
```

- intuitivamente, **@RestController** è un tipo di **@Controller** che implica **@ResponseBody** per le sue operazioni – dunque **luckyWord()** restituisce una stringa e non il nome di una vista

59

Spring Boot

Luca Cabibbo ASW



- Proprietà esterne

- Per completare l'applicazione è necessario specificare il valore della proprietà **lucky.word** – ci sono diversi modi in Spring Boot per specificare una proprietà, tra cui
 - mediante argomenti della linea di comando
 - ad es., `java -jar lucky-word.jar --lucky.word=Happy`
 - mediante proprietà di sistema della JVM
 - `java -jar -Dlucky.word=Happy lucky-word.jar`
 - mediante variabili d'ambiente del sistema operativo
 - `LUCKY_WORD="Happy" java -jar lucky-word.jar`
 - con Gradle: `LUCKY_WORD="Happy" gradle bootRun`
 - mediante un file di proprietà **application.properties** o **application.yml** esterno all'applicazione o archiviato insieme all'applicazione (discusso dopo)
 - mediante una sorgente di proprietà specificata con **@PropertySource**

60

Spring Boot

Luca Cabibbo ASW



Proprietà esterne

- **SpringApplication** carica le proprietà da un file di proprietà **application.properties** oppure da un file YAML **application.yml**

- esempio di file **application.properties**

```
# application.properties
lucky.word=Happy
```

- esempio di file **application.yml** – YAML è un linguaggio di markup, che usa una strutturazione gerarchica basata sull'indentazione dei nomi

```
# application.yml
lucky:
  word: Happy
```



Formati .properties e .yml

- Questi ulteriori esempi mostrano le differenze nelle sintassi di **application.properties** e di **application.yml**

- esempio di file **application.properties**

```
# application.properties
spring.datasource.url=jdbc:postgresql://localhost:5432/restaurantDB
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
```

- esempio di file **application.yml**

```
# application.yml
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/restaurantDB
    username: dbuser
    password: dbpass
  jpa:
    database-platform: org.hibernate.dialect.PostgreSQLDialect
```



- Profili di configurazione

- Alcune applicazioni devono poter essere rilasciate in ambienti di esecuzione diversi (ad es., uno di test e uno di produzione) – e nei diversi casi con una configurazione un po' differente
 - questo scenario può essere gestito mediante i **profili** di configurazione
 - un'applicazione può avere più profili
 - i file **application.properties** o **application.yml** possono specificare le proprietà comuni a tutti i profili
 - per ogni profilo **profile** dell'applicazione si può definire un file di proprietà aggiuntive **application-profile.properties** o **application-profile.yml**
 - il formato YAML consente anche di specificare più profili in un singolo file
 - il profilo attivo di un'applicazione è specificato dalla proprietà **spring.profiles.active** (discussa dopo)



Profili di configurazione

- Un esempio di file di configurazione multi-profilo con YAML
 - i profili sono separati da **---** e i loro nomi specificati dalla proprietà **spring.config.activate.on-profile**

```
# application.yml

---
# questo e' il profilo di default:
lucky:
  word: Default

---
spring:
  config.activate.on-profile: english
lucky:
  word: Happy

---
spring:
  config.activate.on-profile: italian
lucky:
  word: Evviva
```




Selezione del profilo

- Il profilo attivo di un'applicazione è specificato dalla proprietà `spring.profiles.active` – ci sono diversi modi per selezionare il profilo attivo di un'applicazione
 - mediante argomenti della linea di comando
 - ad es., `java -jar lucky-word.jar --spring.profiles.active=english`
 - mediante proprietà di sistema della JVM
 - `java -jar -Dspring.profiles.active=english lucky-word.jar`
 - mediante variabili d'ambiente del sistema operativo
 - `SPRING_PROFILES_ACTIVE=english`
`java -jar lucky-word.jar`
 - oppure `SPRING_PROFILES_ACTIVE=english gradle BootRun`



* Discussione

- Spring Boot ha lo scopo di semplificare lo sviluppo e l'esecuzione di applicazioni Spring
 - codice e configurazioni minimali
 - dipendenze starter
 - configurazione automatica delle applicazioni
 - un approccio basato su convenzioni (“opinionated”)
 - Spring Web MVC – applicazioni web (JPS, Thymeleaf e altro)
 - le applicazioni (web) possono essere eseguite come JAR o WAR, embedded oppure in un proprio application server
 - Spring Data JPA – repository dinamici
 - Spring Boot Actuator – gestione e monitoraggio remoto delle applicazioni
 - configurazioni basate su proprietà e profili
 - vedremo ulteriori caratteristiche in dispense successive