Spring boot

Jacopo De Angelis

5 marzo 2020

Indice

To do

- Finire Spring in action
- Finire Mastering Spring
- Terminare corso Spring core

Capitolo 1

Spring in action

Capitolo 2

Mastering Spring 5.0

IDE per Spring.
Windows builder
Sring webmvc.
Spring, versione web
Spring framework completo Comprende web, workflow Vantaggi di Spring:

- Unit testing semplificato: Spring ha portato il cocetto della dependency injection.
- Riduzione del codice di routine: Spring prende per assodate certe parti di codice. Questo è possibile perchè Spring usa un altro paradigma di programmazione: la programmazione orientata agli aspetti (AOP). il jdbcTemplate può essere creato nel contesto di Spring e "iniettato" (injected) all'interno dei DAO quando serve.
- Flessibilità architetturale: Spring è modulare e, soprattutto, non si propone come unica soluzione, infatti offre integrazioni anche con altri framework.

```
// JAVA BASE

PreparedStatement st = null;
try {
  st = conn.prepareStatement(INSERT_TODO_QUERY);
```

```
  st.setString(1, bean.getDescription());
  st.setBoolean(2, bean.isDone());
  st.execute();
}catch (SQLException e) {
  logger.error("Failed : " + INSERT_TODO_QUERY, e);
} finally {
  if (st != null) {
    try {
      st.close();
    } catch (SQLException e) {
    // Ignore - nothing to do..
    }
  }
}
-------------------------------------------
// JAVA SPRING

jdbcTemplate.update(INSERT_TODO_QUERY,
bean.getDescription(), bean.isDone());
```

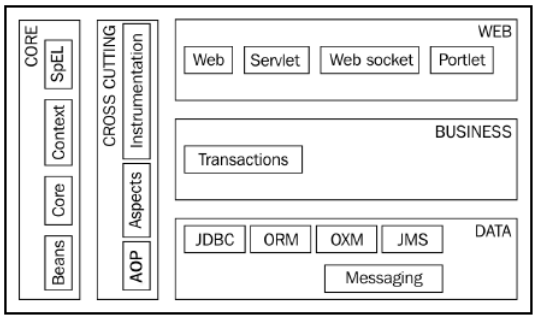

Figura 2.1: Moduli di Spring

Moduli La modularità di Spring è uno dei suoi aspetti più importanti. La figura 2.1 mostra i vari moduli, organizzati in base al loro layer di applicazione. I principali aspetti di Spring sono:

- IoC: non creare oggetti, delegalo al framework

- Dependency injection:
-
-
-
-

Spring Core Container Fornisce la feature al centro di Spring, la dependency injection, tramite il container IOC (inversion of control)(paragrafo 2.1). I moduli del core di spring più importanti sono:

Moduli/artefatti	Uso
spring-core	Utilities usate dagli altri moduli di Spring
spring-beans	Supporto per i beans. In combinazione con spring-core offre la funzionalità base del framework, la dependency injection. Include l'implementazione di una bean factory.
spring-context	Implementa ApplicationContext, che estende BeanFacotry e offre supporto al caricamento delle risorse e alla localizzazione, tra le varie cose.
spring-expression	Estende EL (Expression Language da JSP) e offre un linguaggio per l'accesos e la manipolazione dei beans

Figura 2.2: Spring core

Problemi trasversali Certi problemi sono applicabili a tutti i livelli: log in, sicurezza, ecc. AOP è solitamente utilizzato per questi problemi.

Moduli/artefatti	Uso
spring-aop	Offre il supporto di base all'AOP
spring-aspect	Offre le integrazioni con il framework AOP più usato, AspectJ
spring-instrument	Offre il supporto di base
spring-test	Offre il supporto di base per il unit testing e l'integration testing

Figura 2.3: Moduli per i problemi trasversali

Web Spring offre il suo framework MVC, Spring MVC. Due dei moduli più importanti sono:

- spring-web
- spring-webmvc

Business Il livello di business si concentra sull'eseguire le operazioni logiche dell'applicazione. Solitamente vengono implementate tramite dei POJO (Plain Old Java Object). Le transazioni di Spring (spring-tx) offrono una gestione transazionale dichiarativa per queste classi.

Data Solitamente il livello dati para al database e alle interfacce esterne.

Moduli/artefatti	Uso
spring-aop	Offre il supporto di base all'AOP
spring-aspect	Offre le integrazioni con il framework AOP più usato, AspectJ
spring-instrument	Offre il supporto di base
spring-test	Offre il supporto di base per il unit testing e l'integration testing

Figura 2.4: Moduli per i dati

2.1 Capitolo 2:Dependency injection

[1] Spring si occupa di creare e collegare gli oggetti grazie all'IoC container. Spring può "iniettare" dipendenze: immaginiamo di avere Interfaccia1, implementatta da B1, B2. Se A crea una new su Interfaccia1, spring la implementerà come B1 o B2 in base a ciò che serve.

2.1.1 Comprendere la dependency injection

Ora, immaginiamo di dover integrare all'interno di un servizio di business che richiede un qualche tipo di ordinamento, ad esempio bubble sort. Abbiamo due opzioni:

[1]Non c'è il capitolo 1 poichè era solo di introduzione

1. Inserire all'interno della classe il codice dell'algoritmo
2. Inserirlo in un'interfaccia, collegarla alla classe e successivamente collegarlo.

Nel primo caso, in caso di cambiamenti all'algoritmo, avremo bisogno di modificare il codice per ogni classe che lo implementa, nel secondo caso dovremo modificarlo una sola volta.

A questo punto potremmo anche non far creare a BusinessServiceImpl un'istanza di DataServiceImpl, potremmo invece istanziarlo da un'altra parte per poi darlo a BusinessServiceImpl. Dobbiamo impostare quindi un setter per DataService

```
public class BusinessServiceImpl {
  private DataService dataService;

  public void setDataService(DataService
      ↪ dataService) {
    this.dataService = dataService;
  }

  public long calculateSum(User user) {
    long sum = 0;
    for (Data data : dataService.retrieveData(user)) {
      sum += data.getValue();
    }
    return sum;
  }
}
```

Ora BusinessServiceImpl può lavorare con qualsiasi implementazione di DataService.

Ora, chi decide che istanza creare per DataServiceImpl? Ci pensa l'IOC container.

2.1.2 IOC container

3 domande compaiono:

1. Come fa l'IOC container a sapere quale bean creare per le istanze?
2. Come sa come collegare i bean assieme, ovvero come sa di dover "iniettare" l'istanza di DataserviceImpl in BusinessServiceImpl?
3. Come sa dove cercare i bean?

Definire i bean e wiring

Iniziamo con la prima domanda:

"Come fa l'IOC container a sapere quale bean creare per le istanze?"

Dobbiamo dire all'IOC container che bean creare. Possiamo farlo tramite le annotazioni @Repository, @Component, @Service sulle classi per le quali i bean devono essere creati. Queste annotazioni indicano a Spring di creare i bean per le classi specificate.

@Component è la maniera più generica per definire un bean. Le altre devono essere più specifiche per il contesto nel quale vengono usate. **@Service** è usata nei componenti di business, **@Repository** nei DAO.

Usiamo l'annotazione @Repository in DataServiceImpl perchè è legata all'accesso dati da un database (DAO). Usiamo @Service su BusinessServiceImpl perchè è un servizio di business.

```
@Repository
public class DataServiceImpl implements DataService

@Service
public class BusinessServiceImpl implements
    ↪ BusinessService
```

Ora, prendiamo la seconda domanda:

"Come sa come collegare i bean assieme, ovvero come sa di dover "iniettare" l'istanza di DataserviceImpl in BusinessServiceImpl?"

Il bean di DataServiceImpl deve essere iniettato all'interno di BusinessServiceImpl. Possiamo farlo tramite l'annotazione **@Autowired** sulla variabile DataService nella classe BusinessServiceImpl.

```
public class BusinessServiceImpl {
@Autowired
private DataService dataService;
```

Creare un IOC container

Ci sono due tipi di container IOC:

- Bean factory: si occupa di tutte le funzionalità di base: il ciclo di vita dei bean e il loro wiring
- Application context: è un superset della bean factory con le funzionalità aggiuntive solitamente necessitate in un contesto enterprise. Viene consigliato di usare sempre questo a meno che non si lavori in un contesto dove l'ottimizzazione della memoria occupata non sia critica.

Per l'application contest possiamo usare una configurazione Java o una configurazione XML. Iniziamo con quella Java.

Configurazione Java per l'application context

```
@Configuration
class SpringContext {...}
```

Questo è un semplice esempio su come creare una configurazione di contesto Java. La chiave è l'annotazione **@Configuration**. Rimane sempre la domanda "Come fa l'IOC container a sapere dove cercare i beans?".

Dobbiamo dire al container il package da cercare definendo una ricerca per le componenti. Aggiungiamo una nuova annotazione, **@ComponentScan**.

```
@Configuration
@ComponentScan(basePackages = {
    ↪ "com.mastering.spring" })
```

```
class SpringContext {...}
```

Abbiamo ora definito la ricerca delle componenti all'interno del package com.mastering.spring.

Con ciò che abbiamo scritto fino ad ora, quando viene lanciato il contesto Spring:

- Viene cercato com.mastering.spring e vengono trovati al suo interno BusinessServiceImpl e DataServiceImpl
- DataServiceImpl non ha dipendenze, viene quindi creato il bean
- BusinessServiceImpl ha dipendenze su DataService. DataServiceImpl è un'implementazione dell'interfaccia DataService, fa quindi matching sul criterio AutoWired. Quindi un bean BusinessServiceImpl viene creato e il bean creato per DataServiceImpl è AutoWired attraverso il setter..

Lanciare l'application context con la configurazione Java

```
public class LaunchJavaContext {
  private static final User DUMMY_USER = new
      ↪ User("dummy");
  public static Logger logger =
    Logger.getLogger(LaunchJavaContext.class);

  public static void main(String[] args) {
    ApplicationContext context =
      new AnnotationConfigApplicationContext(
      SpringContext.class);

    BusinessService service =
      context.getBean(BusinessService.class);
    logger.debug(service.calculateSum(DUMMY_USER));
  }
}
```

"ApplicationContext context = new AnnotationConfigApplicationContext(SpringContext.class);" si occupa di creare l'application context. Una votla che il context è partito, dobbiamo ricevere

il bean del BusinessService. Usiamo il metodo getBean che passa il tipo di bean come argomento (BusinessService.class):

```
BusinessService service
  =
context.getBean(BusinessService.class );
```

Configurazione XML per l'application context

Definire la configurazione Spring con XML Semplicemente si crea un file XML, in questo caso chiamato BusinessApplicationContext.xml, e si inserisce **"context: component-scan"**.

```
<?xml version="1.0" encoding="UTF-8"
    ↪ standalone="no"?>
<beans> <!-Namespace definitions removed-->
  <context:component-scan base-package =
    "com.mastering.spring"/>
</beans>
```

Lanciare l'application context con la configurazione XML

```
public class LaunchXmlContext {
  private static final User DUMMY_USER =
    new User("dummy");
  public static Logger logger =
    Logger.getLogger(LaunchJavaContext.class);

  public static void main(String[] args) {
    ApplicationContext context = new
      ClassPathXmlApplicationContext(
      "BusinessApplicationContext.xml");
    BusinessService service =
      context.getBean(BusinessService.class);
    logger.debug(service.calculateSum(DUMMY_USER));
  }
}
```

"ApplicationContext context = new ClassPathXmlApplication-Context("BusinessApplicationContext.xml");" si occupa di creare l'application context.

2.1.3 Tipi di dependency injection

Setter injection Si usano i metodi setter. Ad esempio:

```
public class BusinessServiceImpl {
  private DataService dataService;

  @Autowired
  public void setDataService
    (DataService dataService) {
    this.dataService = dataService;
  }
}

//_____________Altrimenti_____________________

public class BusinessServiceImpl {
  @Autowired
  private DataService dataService;
}
```

Anche se, nella realtà, per usare una setter injection non serve nemmeno definire un metodo setter. Se si specifica @Autowired sulla variabile, Spring userà automaticamente la setter injection.

Costructor injection

```
public class BusinessServiceImpl {
  private DataService dataService;

  @Autowired
  public BusinessServiceImpl
    (DataService dataService) {
    super();
    this.dataService = dataService;
  }
}
```

Quando viene lanciato il codice si vedrà nel log una frase che spiega come l'autowiring sia avvenuto nel costruttore.

Setter vs constructor Solitamente la constructor injection veniva usata per le dipendenze obbligatorie, la setter per quelle facoltative. Bisogna però notare che quanto si usa @Autowired su di un campo o un metodo, la dipendenza è richiesta di default. Se non ci sono candidati all'autowire viene lanciata un'ipotesi. La scelta non è più così chiara quindi.

2.1.4 Lo scope degli Spring beans

I bean di spring pososno essere creati con multipli scope, singleton è quello di default. Lo scope può essere definito tramite l'annotazione **@Scope** su qualsiasi bean.

```
@Service
@Scope("singleton")
public class BusinessServiceImpl implements
    ↪ BusinessService
```

Scope	Uso
singleton	Solo un'istanza di questo bean è usata per istanza dell'IOC container. Anche se ci sono multiple referenze al bean, viene creato solo una volta per container. La singola istanza è salvata e usata per tutte le richiesta successive. Notare che in base a questa definizione, se ci fossero più Container ci potrebbero essere più istanze. Per questo il singleton di Spring è leggermente diverso dal classico.
prototype	Una nuova istanza è creata ogni volta che viene richiesta dal container. Se il bean ha uno stato, è raccomandato usare prototype.
request	Utilizzabile solo nello Spring web context. Una nuova istanza è creata per ogni richiesta http. Viene scartato quando la richiesta è risolta. Utile per i dati che contengono una singola richiesta.
session	Utilizzabile solo nello Spring web context. Una nuova istanza è creata per ogni sessione http. Ideale per tenere i dati legati ad un singolo utente.
application	Utilizzabile solo nello Spring web context. Un'istanza per ogni applicazione web. Ideale per cose come la configurazione per un determinato ambiente.

Figura 2.5: Tipi di scope

2.1.5 Annotazioni

- **@Controller**:
- **@RequestMapping**: cosa fare quando arriva un url. Deve esserci univocità di invocazione (url + metodo)
- **@ModelAttribute**: mettere sul model la variabile che si sta annotando e il nome sarà il nome della variabile
- **@Valid** (da hybernate): regola di validazione
- **@PathVariable**: associa il valore della variaible nell'url alla risorsa
- **@ResponseBody**: cosi facendo Spring scrive nel body la stringa restituita.
- **@AutoWired**: dependencies injection
- **@Primary**: quando è usata su di un bean, diventa il primo ad essere usato se ci sono più candidati.
- **@Qualifier**: serve ad indicare a Spring con quale bean specifico istanziare. Qua serve una spiegazione ulteriore[2]
- **@PreDestroy**: Il metodo è chiamato prima che un bean venga rimosso dal container, rilascia tutte le risorse usate dal bean

Si inizia dal front controller (web.xml), tramite base package si vede dove cercare il controller d'ingresso, per sapere quando chiamare il controller si vede il mapping su cosa è eseguito.

@Qualifier

Qualifer serve a specificare a Spring quale classe istanziare. Bisogna associare ad ogni classe figlia il tag @Qualifier(Stringa di riconoscimento). Lo stesso tag verrà scritto sopra l'istanziazione per specificare il bean da usare.

[2]Vedi paragrafo 2.1.5

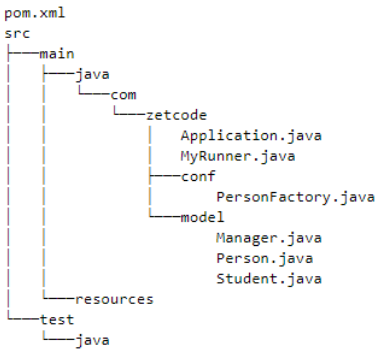

Figura 2.6: Struttura classi

```java
// Person.java
package com.zetcode.model;

public interface Person {

    String info();
}
//---------------------------
// Student.java
package com.zetcode.model;

import
    ↪ org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component
@Qualifier("student")
public class Student implements Person {

    @Override
    public String info() {

        return "Student";
    }
}
//---------------------------
```

```
// Manager.java
package com.zetcode.model;

import
    ↪ org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component
@Qualifier("manager")
public class Manager implements Person {

    @Override
    public String info() {
        return "Manager";
    }
}
//_____________________________
// myRunner.java
@Component
public class MyRunner implements CommandLineRunner {

    private static final Logger logger =
        ↪ LoggerFactory.getLogger(MyRunner.class);

    @Autowired
    @Qualifier("student")// Speicifico che p1 e' uno
        ↪ studente
    private Person p1;

    @Autowired
    @Qualifier("manager") // Specifico che p2 e' un
        ↪ manager
    private Person p2;

    @Override
    public void run(String... args) throws Exception {

        logger.info("{}", p1.info());
        logger.info("{}", p2.info());
    }
}
```

@Autowired

Autowired può essere usato su proprietà, setter e costruttori.

Autowired e proprietà

```
@Component("fooFormatter")
public class FooFormatter {

    public String format() {
        return "foo";
    }
}
// ______________________________

@Component
public class FooService {

    @Autowired
    private FooFormatter fooFormatter;

}
```

Quando FooService viene creato, Spring genera automaticamente un FooFormatter e lo lega alla sua variabile

Autowired e setter

```
public class FooService {

    private FooFormatter fooFormatter;

    @Autowired
    public void setFooFormatter(FooFormatter
        ↪ fooFormatter) {
            this.fooFormatter = fooFormatter;
    }
```

```
}
```

Quando FooService è creato viene automaticamente chiamato il metodo setFooFormatter con l'istanza di FooFormatter.

Autowired e costruttori

```
public class FooService {

    private FooFormatter fooFormatter;

    @Autowired
    public FooService(FooFormatter fooFormatter) {
        this.fooFormatter = fooFormatter;
    }
}
```

QUando FooService viene ceato, un'istanza di FooFormatter è creata e usata come argomento del costruttore.

@Primary

Quando l'annotazione è utilizzata su di un bean, dievnta il primo ad essere usato nel caso ci siano più candidati.

Annotazioni per la classe

@Component È l'annotazione più generica per definire un bean

@Service Usato per definire le classi del service layer

@Repository Usato per definire i DAO

@Controller Usato per definire i controller[3]

@Resource

Mette assieme Autowired e Qualifier in una sola annotazione con sintassi @Resource(value="[qualifier]").

@PostConstruct

Usato su di un metodo di un bean. Il metodo viene chiamato una volta che il bean è inizializzato con le sue dipendenze. Viene chiamato una sola volta durante il ciclo di vita del bean.

@PreDestroy

Usato su di un metodo di un bean. Il metodo viene chiamato prima che il bean venga rimosso dal container. Può essere usato per rilasciare tutte le risorse trattenute dal bean.

2.2 Capitolo 3: Creare un'applicazione web con Spring MVC

Il modello è quello MVC con front controller. In questo modello, i vari controller vengono chiamati da un controller primario che indirizza le richiste in base al contesto. Non vogliamo appesantire i controller con troppe funzioni, per questo c'è il front controller che contatterà i servizi necessari. Alcune responsabilità di un tipico front controller:

1. Decide quale controller esegue la richista
2. decide quale view mostrare

[3]GAC

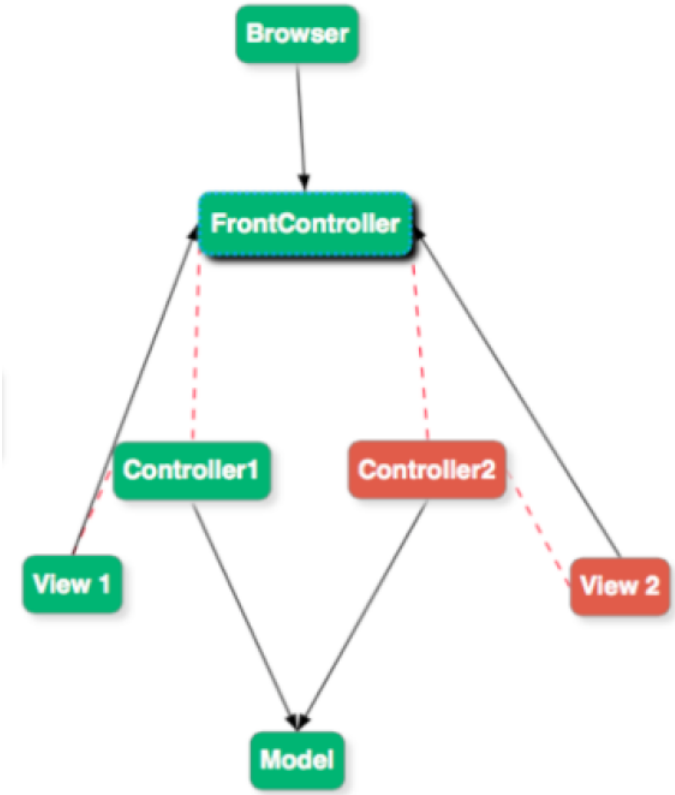

Figura 2.7: MVC con front controller

3. aggiunge funzionalità
4. il front controller di Spring si chiama **DispatcherServlet**

2.2.1 Flow tipico

Vedremo ora 6 tipici flow:

1. Controller senza view: offre direttamente i contenuti (paragrafo 2.2.1)
2. Controller con view (JSP) (paragrafo 2.2.1)
3. Controller con view e con ModelMap (paragrafo 2.2.1)

4. Controller con view e con ModelAndView (paragrafo 2.2.1)
5. Controller per un semplice form (paragrafo 2.2.1)
6. Controller per un semplice form con validazione (paragrafo 2.2.1)

Setup

Prima di tutto dobbiamo impostare l'applicazione per l'uso di Spring MVC. Dobbiamo:

- Aggiungere le dipendenze per SPring MVC
- Aggiungere DispatcherServlet a web.xml
- Creare un application context

Aggiungere le dipendenze per Spring MVC Iniziamo aggiungendo le dipendenze a pom.xml.

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
</dependency>
```

Aggiungere DispatcherServlet a web.xml

Successivamente dobbiamo aggiungere il DispatcherServlet.

```
<servlet>
  <servlet-name>
    spring-mvc-dispatcher-servlet
  </servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
```

```
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/user-web-context.xml
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>
    spring-mvc-dispatcher-servlet
  </servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

La prima parte è per definire una servlet. Viene definita anche una cartella di configurazione, */WEB-INF/user-web-context.xml*. Nella seconda parte stiamo definendo un mapping, ovvero stiamo dicendo che questo controller gestirà tutte le pagine che iniziano con / (*<url-pattern>/</url-pattern>*), quindi tutte.

Creare un application context

```
<beans > <!-Schema Definition removed -->
  <context:component-scan
  base-package="com.mastering.spring.springmvc" />
  <mvc:annotation-driven />
</beans>
```

Con questo definiamo che la ricerca delle componenti sarà all'interno di com.mastering.spring.springmvc, in questo modo tutti i controller e i bean saranno creati e wired.

Usando l'annotazione *<mvc:annotation-driven />* avviamo il supporto per una serie di feature come:

- Mapping
- Gestione delle eccezioni

- Data binding e validazione
- Conversione automatica quando viene usata l'annotazione @RequestBody

Flow 1 - Controller senza view

Iniziamo con un controller semplice:

```
@Controller
public class BasicController {
  @RequestMapping(value = "/welcome")
  @ResponseBody
  public String welcome() {
    return "Welcome to Spring MVC";
  }
}
```

Parole chiave:

- **@Controller**: definisce un controller che può contenere dei @RequestMapping
- **@RequestMapping(value = "/welcome")**: Definisce un mapping dell'URL legato al metodo welcome. Quando nel broser si accede alla sottopagina /welcome, Spring MVC esegue il metodo.
- **@ResponseBody**: In questo caso specifico, l'output del metodo viene mandato al browser come contenuto della risposta. La funzione di ResponseBody non si limita a questo però

Flow 2 - Controller con view (JSP)

Nell'esempio precedente il testo da mostrare sul browser era all'interno del controller, pessima scelta dal punto di vista delle good practice. Generalemten il contenuto delle pagine è generato

tramite View. Solitamente vengono usate le JSP, Java Server Pages.

```
@Controller
public class BasicViewController {
  @RequestMapping(value = "/welcome-view")
  public String welcome() {
    return "welcome";
  }
}
```

Flow 3 - Controller con view e con ModelMap

Generalmente per generare una view dovremmo passarle dei dati. In Spring possiamo passarle i dati usando un modello.

```
@Controller
public class BasicModelMapController {
  @RequestMapping(value = "/welcome-model-map")
  public String welcome(ModelMap model) {
    model.put("name", "XYZ");
    return "welcome-model-map";
  }
}
```

Parole chiave:

- @RequestMapping(value = "/welcome-model-map"): l'URI è mappata su /welcome-model-map
- public String welcome(ModelMap model:Il nuovo parametro agguinto + ModelMap model. Spring creerà un modello e lo renderà disponibile per quesot metodo. Gli attributi del modello saranno utilizzabili nella view
- model.put("name", "XYZ"): Aggiunge al modello un campo "name" con valore "XYZ"

Ora, per creare una vie creiamo una JSP in WEB-INF/views/welcome-model-map.jsp

```
Welcome ${name}! This is coming from a model-map - a
    ↪ JSP
```

$name usa la sintassi EL (Expression Language) per accedere all'attributo del modello.

Flow 4 - Controller con view e con ModelAndView

Spring offre un approccio differente utilizzando ModelAndView. Il controller può ritornare un oggetto ModelAndView con il nome della view e gli appropriati attributi nel Model.

```
@ControllerJava
public class BasicModelViewController {
  @RequestMapping(value = "/welcome-model-view")
  public ModelAndView welcome(ModelMap model) {
    model.put("name", "XYZ");
    return new ModelAndView("welcome-model-view",
        ↪ model);
  }
}
```

Parole chiave:

- @RequestMapping(value = "/welcome-model-view"): l'URI è mappata su /welcome-model-view
- public ModelAndView welcome(ModelMap model): notare che il return non è una stringa ma una ModelAnd View
- return new ModelAndView("welcome-model-view", model): crae un oggetto ModelAndView con le appropriate view e modello.

Ora creiamo la view:

```
Welcome ${name}! This is coming from a model-view -
    ↪ a JSP
```

Flow 5 - Controller per un semplice form

Ora concentriamoci sulla creazione di un semplice form per ricevere un input dall'utente. Si richiede:

- Creare un semplice POJO, il bean dell'utente
- Creare una coppia di controller: uno per mostare il form e uno per catturare i dettagli inseriti in esso
- Creare una semplice view

I POJO sono generalmente usati per rappresentare i bean seguendo la tipica notazione JavaBean. Contengono le variabili private con i metodi getter e setter e il costruttore senza argomenti.

```
public class User {
  private String guid;
  private String name;
  private String userId;
  private String password;
  private String password2;

  //Constructor
  //Getters and Setters
  //toString
}
```

Alcune cose importanti:

- Questa classe non ha annotazioni di Spring
- Cattureremo nome, UserId e password nel form. abbiamo un campo di conferma password, pawwrod2, e un identificatore unico guid
- Costruttori, getter e setter e toString non sono mostrati solo per brevità

Controller:

```
@Controller
public class UserController {
  private Log logger = LogFactory.getLog
    (UserController.class);

  @RequestMapping(value = "/create-user", method =
      ↪ RequestMethod.GET)
  public String showCreateUserPage(ModelMap model) {
    model.addAttribute("user", new User());
    return "user";
  }
}
```

Punti chiave:

- @RequestMapping(value = "/create-user", method = RequestMethod.GET): stiamo mappando l'URI su /create-user. Per la prima volta abbiamo specificato una *request* usando l'attributo method. QUesto metodo verrà invocato solo per le richieste HTTP GET. Non verrà invocato per altre azioni, tipo POST.
- public String showCreateUserPage(ModelMap model): tipico metodo di controllo
- model.addAtribute("user", new User()): per creare il modello con un bean vuoto dietro

Ora, le Java Server Pages (JSP)sono una delle tecnologie di Spring per supportare le view. Spring rede più semplice la creazione di views offrendo una libreria di tag. Qusti includono i tag per vari elementi dei form, binding, validazione, setting e messaggi multilingua.

Ora creiamo il file /WEB-INF/views/user.jsp. Prima di tutto creiamo le reference alle librerie da usare:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
    ↪ prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt"
    ↪ prefix="fmt"%>
```

```
<%@ taglib
    ↪ uri="http://www.springframework.org/tags/form"
    ↪ prefix="form"%>
<%@ taglib uri="http://www.springframework.org/tags"
    ↪ prefix="spring"%>
```

Le prime due entry sono per JSTL e per formattare le librerie di tag. Usiamo un prefisso (*prefix*) per agire da scorciatoia per l'uso dei tag. Ora creiamo il form:

```
<form:form method="post" modelAttribute="user">
  <fieldset>
    <form:label path="name">Name</form:label>
    <form:input path="name"
      type="text" required="required" />
  </fieldset>
</form:form>
```

Parole chiave:

- <form:form method="post" modelAttribute="user">: Il tag form della libreria Spring con due attributi specificati: il metodo di invio (POST) e il tipo di modello per la modellazione degli attributi dell'oggetto (user).
- <fieldset>: elemento HML per raggruppare una serie di controlli
- <form:label path="name">Name</form:label>: tag Spring per mostrare un'etichetta. Il apth specifica il nome del campo del bean al quale viene applicata questa etichetta
- <form:input path="name" type="text" required="required" />: Questo è il tag di Spring per creare un campo di input di testo. Il path specifica il campo del nome nel bean sul quale è mappato.

Quando usiamo il tag "form" di Spring, l'oggetto di backup (modelAttribute = "user) è collegato direttamente al form e inviando il form i valori vengono passati direttamente all'oggetto.

Per fare un esempio più completo:

```
<form:form method="post" modelAttribute="user">
<form:hidden path="guid" />
<fieldset>
  <form:label path="name">Name</form:label>
  <form:input path="name"
    type="text" required="required" />
</fieldset>
<fieldset>
  <form:label path="userId">User Id</form:label>
  <form:input path="userId"
    type="text" required="required" />
</fieldset>
<!-password and password2 fields not shown for
    ↪ brewity-->
<input class="btn btn-success" type="submit"
    ↪ value="Submit" />
</form:form>
```

Quando l'utente fa il submit, il browser invoca una richiesta HTTP POST. Ora vedremo come creare un metodo per gestirla. Per rendere le cose semplici faremo anche un log dell'oggetto per mostrarlo direttamente.

```
@RequestMapping(value = "/create-user", method =
    ↪ RequestMethod.POST)
public String addTodo(User user) {
  logger.info("user details " + user);
  return "redirect:list-users";
}
```

Parole chiave:

- @RequestMapping(value = "/create-user", method = RequestMethod.POST): Poichè vogliamo gestire il submit del form usiamo il metodo RequestMethod.POST, così come indicato nel form
- public String addTodo(User user): Usiamo l'oggetto di backing del form come parametro. Spring MVC saprà automaticamente collegare i campi

- logger.info("user details " + user): viene loggato tutto
- return redirect:list-users: solitamente, una volta aggiornato il database, si reindirizza l'utente ad un'altra pagina. in questo caso a /list-user. QUando usiamo redirect Spring invia una HTTP response con status 302 che segnala un redirect ad una nuova URL

Il codice per mostrare tutti gli utenti è molto semplice:

```
@RequestMapping(value = "/list-users", method =
    ↪ RequestMethod.GET)
public String showAllUsers() {
  return "list-users";
}
```

Flow 6 - Controller per un semplice form con validazione

Nel precedente form non validavamo i valori. Mentre possiamo scrivere del codice Javascript per validarli (e per farci del male), è sempre meglio validare i dati a livello server.

Spring offre un'ottima implementazione con Bean Validation API, gestita tramite l'Hibernate validator.

Iniziamo aggiungendo il validatore al pom.xml:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>5.0.2.Final</version>
</dependency>
```

E ora la magia, per semplicemente validare i campi basta aggiungere agli attributi del bean le annotazioni collegate:

```
@Size(min = 6,
  message = "Enter at least 6 characters")
private String name;
```

```
@Size(min = 6,
  message = "Enter at least 6 characters")
private String userId;

@Size(min = 8,
  message = "Enter at least 8 characters")
private String password;

@Size(min = 8,
  message = "Enter at least 8 characters")
private String password2;
```

"message" è ciò che viene mostrato se la validazione non avviene.
Altre annotazioni per la validazione sono:

- **@NotNull**: It should not be null
- **@Size(min =5, max = 50)**: Maximum size of 50 characters and minimum of 5 characters.
- **@Past**: Should be a date in the past
- **@Future**: Should be a future date
- **@Pattern**: Should match the provided regular expression
- **@Max**: Maximum value for the field
- **@Min**: Minimum value for the field

Vedere la lista completa nella documentazione di jboss.

Ora, per scrivere nel controller di validare i campi, scriviamo:

```
@RequestMapping(value =
    ↪ "/create-user-with-validation", method =
    ↪ RequestMethod.POST)
public String addTodo(@Valid User user,
    ↪ BindingResult result) {
  if (result.hasErrors()) {
    return "user";
  }
```

```
  logger.info("user details " + user);
  return "redirect:list-users";
}
```

Punti chiave:

- public String addTodo(@Valid User user, BindingResult result): QUando l'annotazione @Valid è usata, SPring valida il bean. Il risultato è reso disponibile tramite il BindingResult
- if (result.hasErrors()): Controlla se ci sono errori di validazione
- return "user": se ci sono errori di validazione l'utente viene rimandato alla pagina.

A questo punto dobbiamo potenziare la user.jsp per mostrare i messaggi di validazione. QUesto è un esempio, gli altri sono simili:

```
<fieldset>
  <form:label path="name">Name</form:label>
  <form:input path="name" type="text"
     ↪ required="required" />
  <form:errors path="name" cssClass="text-warning"/>
</fieldset>
```

<form:errors path="name" cssClass="text-warning"/> è il tag di spring per mostrare gli errori relativi al campo nel path. Possiamo anche legare del CSS per mostrare meglio gli errori.

Se si vuole creare una validazione più complicata per motivi specifici si può usare l'annotazione **@AssertTrue**.

```
@AssertTrue(message = "Password fields don't match")
private boolean isValid() {
  return this.password.equals(this.password2);
}
```

@AssertTrue(message = "Password fields don't match") mostra il messaggio nel caso la validazione non sia effettuata.

2.2.2 Come funziona Spring

Nella sezione precedente abbiamo visto alcune delle feature più importanti:

- Architettura a bassa dipendenza con ruoli indipendenti e ben definiti per ogni oggetto
- Controller altamente flessibile. I metodi del controller possono avere un vario range di parametri e valori di ritorno
- Permette il riuso degli oggetti come oggetti di backup collegati ai form frontend. Riduce il bisogno di avere form separati
- Tag built-in con supporto per la localizzazione
- Il modello usa una hashmap, permette un'integrazione con multiple tecnologie
- Binding flessibile. Gli errori durante la creazione possono essere gestiti come errori di validazione
- Mock MVC Framework per lo unit testing

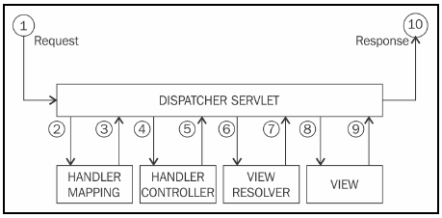

Figura 2.8: Architettura Spring MVC

Ora prendiamo un flow di esempio e valutiamo i passi di esecuzione. Prendiamo il flow numero 4 (pagina 33. L'URL è http://localhost:8080/welcome-model-view. Gli step sno:

1. Il browser manda una richiesta per l'URL. La DispatcherServlet è il front controller e gestisce tutte le richieste. Riceve la richiesta

2. La servlet controlla la URI (/welcome-model-view) e deve identificare i giusto controller per gestirla. Per trovare il controller giusto parla all'handler mapping
3. L'handler mapping ritorna il metodo specifico (in questo esempio welcome in BasicModelViewController) che gestisce la richiesta
4. La DispatcherServlet invoca il metodo specifico dell'handler (public ModelAndView welcome(ModelMap model)).
5. Il metodo ritorna il modello e la view. In questo esempio un oggetto ModelAndView.
6. DIsatcherServlet ha il nome logico della view (da ModelAndView *welcome-model-view*). Ora deve capire come determinare il nome fisico della view. Controlla se ci sono dei view resolver disponibili. Trova il view resolver configurato (org.springframework.web.servlet.view.InternalResourceViewResolver). Chiama il resolver, dandogli il nome logico come input.
7. Il resolver esegue la logica per mappare il nome logico sul nome fisico. In questo caso welcome-model-view è tradotto in /WEB-INF/views/welcome-model-view.jsp
8. Il DIspatcherServlet esegue la view. rende anche il modello disponibile alla view
9. La view ritorna il contenuto per essere inviato alla DispatcherServlet
10. La DispatcherServlet invia la risposta al browes

Ora possiamo effettivamente capire alcuni concetti importanti di Spring.

RequestMapping

@RequestMapping è usato per mappare una URI ad un controller o ad un metodo del controller. Un parametro aggiuntivo, RequestMethod, permette di mappare il metodo ad una specifica richesta (POST, GET, ecc.)

Metodi accettati sugli argomenti dei metodi con RequestMapping:

- java.util.Map/org.springframework.ui.Model/org.springframework.ui.ModelMap: agisce come modello che conterra i valori che sono esposti alla view
- oggetti form o comandi: usati per legare le richieste ai parametri dei bean. Supportano la validazione
- org.springframework.validation.Errors/org.springframework.validation.BindingResult: risultato della validazione
- @PreDestroy: su ogni bean, questo metodo è chiamato prima che un bean venga rimosso dal container. Può essere usato per rilasciare le risorse bloccate dal bean
- @RequestParam: L'annotazione per accedere ad un parametro HTTP specifico
- @RequestHeader: l'annotazione per accedere ad un header HTTP specifico
- @SessionAttribute: l'annotazione per accedere ad un attributo specifico della session
- @RequestAttribute: l'annotazione per accedere ad un attributo specifico della richiesta HTTP
- @PathVariabile: l'annotazione che permette l'accesso alle variabili dell'URI. Capiremo meglio coi microservizi

@RequestMapping supporta una varietà di tipi di ritorno. Pensando concettualmente, un metodo con RequestMapping dovrebbe rispondere a due domande:

1. Qual è la view?
2. Qual è il modello che la view necessita?

Nonostante ciò, con Spring, view e model possono non essere dichiarati esplicitamente tutte le volte:

- Se una view non è esplicitamente definita come tipo di ritorno, allora è definita implicitamente
- Similmente, qualsiasi oggetto è sempre arricchieto come descritto più sotto

Spring usa una semplice regola per determinare le esatte view e model. Una coppia di regole importanti:

- **Arricchimento implicito del modello**: se un modello fa parte del tipo di ritorno, è arricchito con o dei comandi (inclusi i risultati dalle validazioni). In aggiunta, i risultati dei metodi con l'annotazione @ModelAttributte sono aggiunti al modello.
- **Implicita determinazione della view**: se il nome della view non è nel tipo di ritorno, è determinata usando DefaultRequestToViewNameTranslator. Di default rimuove lo slash di inizio e di fine così come l'estensione del file dall'URI (ad esempio display.html diventa display)

I tipi di ritorno che sono supportati sui metodi del Controller con RequestMapping:

- **ModelAndView**: l'oggetto include un puntatore al modello e al nome della view
- **Model**: solo il modello viene ritornato, il nome della view è determinato tramite il DefaultRequestToViewNameTranslator
- **Map**: Una semplice map è usata per esporre un modello
- **View**:una view con un modello implicitamente definito
- **String**: puntatore al nome della view

View resolution

Spring offre una view resolution flessibile. Offre varie opzioni:

- Integrazione con JSP, freemaker
- Varie strategie di view resolution, alcune delle quali sono:
  - **XmlViewResolver**: Basata su di un file esterno di configurazione XML
  - **ResourceBundleViewResolver**: basata su di un file di proprietà
  - **UrlBasedViewResolver**: mapping diretto del nome logico della view all'URL
  - **ContentNegotiatingViewResolver**: delega ad altri resolver basato sull'header request Accept
- Supporto per il cambio di view resolver con definizione esplicita dell'ordine di preferenza
- Generazione diretta di XML, JSON e Atom usando la content Negotiation

Un esempio di InternalResourceViewResolver è:

```
<bean id="jspViewResolver" class=
  "org.springframework.web.servlet.view.
  InternalResourceViewResolver">
  <property name="viewClass"
    value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

Il nome fisico della view è determinato usando prefisso e suffisso determinati per la logical view usando JstlView.

Configurare Freemarker Prima di tutto configuriamo il bean usato per caricare i template di Freemarker.

```
<bean id="freemarkerConfig"
  class="org.springframework.web.servlet.view.
  freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath"
     ↪ value="/WEBINF/freemarker/"/>
```

```
</bean>
```

Ora ecco il bean per configurare il view resolver

```
<bean id="freemarkerViewResolver"
  class="org.springframeowork.web.servlet.view.freemarker.FreeMarkerViewResolve
  <property name="cache" value="true"/>
  <property name="prefix" value=""/>
  <property name="suffix" value".ftl"/>
</bean>
```

Attributi dei Model

Solitamente i form online contengono una lista di valori dai menù drop down (ad esempio una lista di stati, una lista di titoli ecc.). QUesta lista deve essere resa disponibile al Model e per farlo si usa l'annotazione @ModelAttribute legata ai metodi.

```
@ModelAttribute
public List<State> populateStateList() {
  return stateService.findStates();
}

  // Per piu' attributi

@ModelAttribute
public void populateStateAndCountryList() {
  model.addAttribute(stateService.findStates());
  model.addAttribute(countryService.findCountries());
}
```

Non c'è una limitazione ai metodi che possono essere segnati tramite @ModelAttribute.

I Model attribute possono esseere resi accessibili tra più controller usando Controller Advice.

Attributi della Session

I successivi attributi vengono usati in una singola richiesta. Comunque ci potrebbero essere valori specifici dello user che non verranno condivisi tra le richiesta. Questi valori solitamente sono salvati nell'HTTP session. Spring MVC offre una semplice annotazione a livello di classe o di tipo, @SessionAttributes per specificare questi attributi:

```
@Controller
@SessionAttributes("exampleSessionAttribute")
public class LoginController {
  ...
  model.put("exampleSessionAttribute", sessionValue);
  ...
}
```

Una volta che definiziamo un attributo nell'annotazione, è automaticamente aggiunto alla sessione se lo stesso attributo è aggiunto al Model. Se ad un certo punto nel controller aggiungiamo un attributo con lo stesso nome al model, questo verrà aggiunto anche alla sessione.

Il valore può essere reso accessibile in altri controller specirficando l'annotazione @SessionAttribute a livello di tipo. Il valore sarà reso automaticamente disponibile a tutti e potrà essere accessibile tramite model:

```
@Controller
@SessionAttributes("exampleSessionAttribute")
public class SomeOtherController {
  ...
  Value sessionValue =
      ↪ (Value)model.get("exampleSessionAttribute");
  ...
}
```

È impostante rimuovere dalla sessione i valori che non sono più necessari. Ci sono due modi per rimuoverli:

1. usare removeAttribute accessibile tramile la classe WebRequest

2. usare il metodo cleanUpAttribute

```
// Metodo 1
@RequestMapping(value="/some-method", method =
    ↪ RequestMethod.GET)
public String someMethod(/*Other Parameters*/
  WebRequest request, SessionStatus status) {
  status.setComplete();
  request.removeAttribute("exampleSessionAttribute",
      ↪ WebRequest.SCOPE_SESSION);
  //Other Logic
}

// Metodo 2
@RequestMapping(value = "/some-other-method", method
    ↪ = RequestMethod.GET)
public String someOtherMethod(/*Other Parameters*/
  SessionAttributeStore store, SessionStatus status)
      ↪ {
  status.setComplete();
  store.cleanupAttribute(request,
      ↪ "exampleSessionAttribute");
  //Other Logic
}
```

La personalizzazione del binding viene effettuata tramite @InitBinder. La personalizzazione può essere fatta in uno specifico controller o in più controller usando Handler Advice.

```
@InitBinder
protected void initBinder(WebDataBinder binder) {
  SimpleDateFormat dateFormat = new
      ↪ SimpleDateFormat("dd/MM/yyyy");
  binder.registerCustomEditor(Date.class, new
      ↪ CustomDateEditor(dateFormat, false));
}
```

Questo codice, ad esemio, impone come verrà configurata la data per tutte le sue occorrenze.

Fondamentalmente InitBinder funziona come un preprocessore dei dati inviati al controller.

```
vinder.registerCustomEditor(
  [classe alla quale applicare il metodo],
  [classe da associare per l'editing])
```

2.2.3 Feature avanzate

Gestione delle eccezioni

Con Spring la maggior parte delle eccezioni è resa unchecked. QUesto permette di gestirle in maniera generale.

Due metodi di gestione delle eccezioni sono:

1. Gestione comune a tutti i controller
2. Eccezioni specifiche per un contoller

Gestione comune a tutti i controller Controller advice può essere usato anche per implementare una gestione comune delle eccezioni tra vari controller:

```
@ControllerAdvice
public class ExceptionController {
  private Log logger =
      ↪ LogFactory.getLog(ExceptionController.class);

  @ExceptionHandler(value = Exception.class)
  public ModelAndView handleException
      ↪ (HttpServletRequest request, Exception ex) {

    logger.error("Request " + request.getRequestURL()
        ↪ + " Threw an Exception", ex);
    ModelAndView mav = new ModelAndView();
```

```
    mav.addObject("exception", ex);
    mav.addObject("url", request.getRequestURL());
    mav.setViewName("common/spring-mvc-error");
    return mav;
  }
}
```

Notiamo alcune parti:

- **@ControllerAdvice**: è applicabile a tutti i controller
- **@ExceptionHandler(value = Exception.class)**: ogni metodo con questa annotazione sarà chiamato quando un'eccezione di quel sottotitpo verrà lanciata
- **public ModelAndView handleException(HttpServletRequest request, Exception ex)**: l'eccezione che viene lanciata è injected nella variabile Exception. Il metodo è dichiarato con un tipo di ritorno ModelAndView per permettere al controller di restituire un model con l'eccezione gestita e una view di gestione.
- **mav.addObject("exception", ex)**: aggiungere un'eccezione al model in modo che possa essere visualizzata nella view
- **mav.setViewName("common/spring-mvc-error")**: la view dell'eccezione

E la view di errore:

```
<%@ taglib prefix="c"
    ↪ uri="http://java.sun.com/jsp/jstl/core"%>
<%@page isErrorPage="true"%>
<h1>Error Page</h1>
URL: ${url}
<br />
Exception: ${exception.message}
<c:forEach items="${exception.stackTrace}"
  var="exceptionStackTrace">
  ${exceptionStackTrace}
</c:forEach>
```

Notiamo che:

- **URL: ${url}**: mostra l'URL del model
- **Exception: ${exception.message}**: mostra il messaggio dell'eccezione. L'eccezione è popolata nel model da ExceptionController
- **forEach around ${exceptionStackTrace}**: mostra la stack trace

Gestione specifica in un Controller Si può lavorare nello stesso modo, semplicemente l'annotazione sarà posta a livello di metodo all'interno del controller.

2.2.4 Internazionalizzazione (I18N)

Può essere implementata usando due approcci:

- SessionLocaleResolver
- CookieLocaleResolver

Nel primo caso, il locale scelto dallo user è salvato nella sessione e, quindi, viene mantenuto solo durante quella sessione. Nell'altro caso invece diventa comune a più sessioni.

Message bundle setup

Prima di tutto, prepariamo il message bundler. Lo snippet è questo:

```
<bean id="messageSource" class=
  "org.springframework.context.support.ReloadableResourceBundleMessageSource">
  <property name="basename"
     ↪ value="classpath:messages" />
  <property name="defaultEncoding" value="UTF-8" />
</bean>
```

Notiamo:

- **class="org.springframework.context.support.ReloadableResourceBundleMessageSource"**: siamo configurando un reloadable resource bundle. Permette il ricaricamento delle proprietà trmaite cache
- *< propertyname = "basename"value = "classpath : messages"/ >*: configura il caricamento delle proprietà tramite messages.properties e il file messages_locale.properties.

Configuriamo una coppia di file di proprietà e rendiamoli accessibili in src/main/resources:

```
// message_en.properties
welcome.caption=Welcome in English

// message_fr.properties
welcome.caption=Bienvenue - Welcome in French
```

Possiamo mostrare i messaggi usando il tag spring:message:

```
<spring:message code="welcome.caption" />
```

Configurare un SessionLocaleResolver

Ci sono due parti del configurare un SessionLocaleResolver:

1. configurare un localeResolver
2. configurare un interceptor per gestire i cambiamenti del locale

```
<bean id="springMVCLocaleResolver"
  class="org.springframework.web.servlet.i18n.SessionLocaleResolver">
  <property name="defaultLocale" value="en" />
</bean>

<mvc:interceptors>
```

```
  <bean id="springMVCLocaleChangeInterceptor"
    class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="language" />
  </bean>
</mvc:interceptors>
```

Notiamo:

- **<property name="defaultLocale" value="en" />**: di default il locale è en
- **<mvc:interceptors>**: LocaleChangeInterceptor è configurato come un HandlerInterceptor. Intercetta tutte le richieste agli handler e controlla il locale
- **<property name="paramName" value="language"/>**: LocaleChangeInterpetor è configurato per usare un parametro dalla HTTP request chiamato language che indica il locale. Quindi qualsiasi URL nel formato *http://server/uri?language=locale* genererebbe un cambio del locale
- Se aggiungi language=en a qualsiasi URL, segnali di usare en per tutta la durata della sessione.

Configurare un CookieLocaleResolver

Usiamo un CookieLocaleResolver così:

```
<bean id="localeResolver"
  class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
  <property name="defaultLocale" value="en" />
  <property name="cookieName"
     ↪ value="userLocaleCookie"/>
  <property name="cookieMaxAge" value="7200"/>
</bean>
```

Notiamo:

- **<property name="cookieName" value="userLocaleCookie"/>**: il nome del cookie salvato nel browser

- **<property name="cookieMaxAge" value="7200"/>**: la durata del cookie è di due ore (7200 secondi)
- poichè stiamo usando un LocaleChangeInterceptor, se si aggiunge language=en a qualsiasi URL, si userebbe il locale en per due ore (o fino a cambiamento)

2.2.5 Integration testing dei controller

Caricando l'intero context per controllare l'integrazione dei controller staremmo parlando di **Integration testing**.

```
@RunWith(SpringRunner.class)
@WebAppConfiguration
@ContextConfiguration("file:src/main/webapp/WEB-INF/user-web-context.xml")
public class BasicControllerSpringConfigurationIT {

  private MockMvc mockMvc;

  @Autowired
  private WebApplicationContext wac;

  @Before
  public void setup() {
    this.mockMvc =
      MockMvcBuilders.webAppContextSetup
        (this.wac).build();
  }

  @Test
  public void basicTest() throws Exception {
    this.mockMvc
      .perform(get("/welcome")
        .accept(MediaType.parseMediaType("application/html;charset=UTF-8")))
      .andExpect(status().isOk())
      .andExpect(content().string("Welcome to Spring
          ↪ MVC"));
  }
}
```

Analizziamolo:

- **@RunWith(SpringRunner.class)**: SpringRunner ci aiuta nel lanciare uno SpringContext
- **@WebAppConfiguration**: Usato per lanciare una web app con Spring MVC
- **@ContextConfiguration("file:src/main/webapp/WEB-INF/user-webcontext. xml")**: specifica il path dell'XML con il contesto di Spring
- **this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build()**: Nei primi esempi abbiamo usato un setup standalone. In questo esempio, invece, lanciamo l'intera web app, per questo usiamo webAppContextSetup
- L'esecuzione è simile agli altri test

2.2.6 Offrire risorse statiche

Il backend, generalmente, viene costruito tramite applicazioni web o servizi REST basati su framework come Spring MVC.

Spring MVC offre alcune feature importanti:

- espongono il contenuto statico dalle cartelle nel web application root
- abilita il caching
- abilita la compressione Gzip del contenuto statico

Esposizione del contenuto statico Le applicazioni web solitamente hanno molti contenuti statici. Spring MVC offre possibilità per esporre questi contenuti direttamente da cartelle specifiche. d esempio:

```
<mvc:resources
  mapping="/resources/**"
  location="/static-resources/"/>
```

Notare:

- **location="/static-resources/"**: La location indica la cartella all'interno del file war o del classpath che vogliamo esporre come contenuto statico. In questo esempio è all'interno della cartella static-resources partendo dal root. Possiamo anche specificare più cartelle usando valori separati da virgole
- **mapping="/resources/**"**: il mapping specifica l'URI utilizzata per accedere al path. Quindi, ad esempio, un file CSS chiamato app.css all'interno della cartella statica sarebbe usato tramite /resources/app.css

La configurazione completa in Java è:

```
@Configuration
@EnableWebMvc
public class WebConfig extends
    ↪ WebMvcConfigurerAdapter {

  @Override
  public void addResourceHandlers
    (ResourceHandlerRegistry registry) {
    registry
      .addResourceHandler("/static-resources/**")
      .addResourceLocations("/static-resources/");
  }
}
```

Caching

Viene effettuato sia a livello di classe che a livello di XML:

```
// Classe
registry
  .addResourceHandler("/resources/**")
  .addResourceLocations("/static-resources/")
  .setCachePeriod(365 * 24 * 60 * 60);
```

```
<!-- XML -->
<mvc:resources
  mapping="/resources/**"
  location="/static-resources/"
  cache-period="365 * 24 * 60 * 60"/>
```

L'header Cache-Control: max-age=specified-max-age verrà mandato al browser.

Abilitare la compressione GZip

Comprimere una response in maniera semplice è un modo molto semplice per rendere un'applicazione web più veloce. Tutti i browser usano Gzip. La compressione e la decompressione sono trasparenti per il browser.

Semplicemente il browser può specificare se accetta contenuto compresso o no. Se il server lo supporta, può essere usato.

La request Header inviata dal browser è: *Accept-Encoding: gzip, deflate*

La response header inviata dalla web application è: *Content-Encoding: gzip*

Ad esempio:

```
registry
  .addResourceHandler("/resources/**")
  .addResourceLocations("/static-resources/")
  .setCachePeriod(365 * 24 * 60 * 60)
  .resourceChain(true)
  .addResolver(new GzipResourceResolver())
  .addResolver(new PathResourceResolver());
```

Notiamo:

- **resourceChain(true)**: Serve per controllare che sia effettivamente possibile inviare file compressi, altrimenti verrà inviato il file completo. Per questo usiamo la resource chain

- **addResolver(new PathResourceResolver())**: PathResourceResolver è il resolver di base

- **addResolver(new GzipResourceResolver())**: abilita la compressione

2.2.7 Integrare Spring MVC con Bootstrap

WebJars ci permette di gestire le versioni di bootstrap tramite Maven. WebJars sono file jar lato client che contengono JS CSS. Possiamo usare Maven o Gradle per scaricarli e renderli disponibili all'applicazione. Il vantaggio di WebJars è che risolve le dipendenze transitive.

I passi sono:

1. aggiungere Bootstrap Webjar come dipendenza di Maven

2. configurare l'MVC resource handler per usare il contenuto statico da WebJars

3. Usare le risorse di Bootstrap nelle JSP

Aggiungere Bootstrap Webjar come dipendenza di Maven

Semplicemente lo aggiungiamo al file pom.xml:

```
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>x.y.z</version>
</dependency>
```

Configurare l'MVC resource handler per usare il contenuto statico da WebJars

Anche questo passaggio è molto semplice. Nello spring context aggiungiamo:

```
<mvc:resources
  mapping="/webjars/**"
  location="/webjars/"/>
```

Con questa configurazione il ResourceHttpRequestHandler rende disponibile il contenuto del WebJars come contenuto statico.

Usare le risorse di Bootstrap nelle JSP

Possiamo usarle come ogni altro contenuto statico:

```
<script src=
  "webjars/bootstrap/3.3.6/js/bootstrap.min.js">
</script>
<link
  href="webjars/bootstrap/3.3.6/css/bootstrap.min.css"
  rel="stylesheet">
```

2.2.8 Spring Security

L'autenticazione è il processo di stabilire l'identità dello user, verificando che sia chi dice di essere.

L'autorizzazione è controllare che abbia i permessi per svolgere una determinata azione.

Una best practice è di rinforzare l'autenticazione e l'autorizzazione su ogni pagina.

Spring security supporta la maggior parte dei meccanismi di autenticazione:

- Form-based
- LDAP
- JAAS
- basati sui container
- sistemi custom

Proviamo a creare una semplice applicazione con SPring Security. I passi sono:

1. Aggiungere le dipendenza da Spring Security
2. Configurare l'intercettazione di tutte le richieste
3. Configurare Spring Security
4. Aggiungere una funzionalità di logout

Aggiungere le dipendenza da Spring Security

La dipendenza, come sempre, va aggiunta al file pom.xml:

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
</dependency>
```

Configurare l'intercettazione di tutte le richieste

La best practice è quella di validare tutte le richieste HTTP. Bisogna farlo tramite un filtro che intercetti tutte le richieste e le convalidi. Useremo un filtro, DelegatingFilterProxy, che delega ad un bean di Spring FilterChainProxy:

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Configurare Spring Security

Usiamo, come sempre, un file di configurazione:

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends
    ↪ WebSecurityConfigurerAdapter {

  @Autowired
  public void configureGlobalSecurity
      ↪ (AuthenticationManagerBuilder auth) throws
      ↪ Exception {
    auth.inMemoryAuthentication()
      .withUser("firstuser")
      .password("password1")
      .roles("USER", "ADMIN");
  }

  @Override
  protected void configure(HttpSecurity http) throws
      ↪ Exception {
    http.authorizeRequests()
      .antMatchers("/login")
      .permitAll()
      .antMatchers("/*secure*/**")
      .access("hasRole('USER')")
```

```
        .and().formLogin();
    }
}
```

Notiamo:

- **@EnableWebSecurity**: L'annotazione consente ad ogni classe di configurazione di contenere la configurazione di Spring. In questo caso, facciamo l'override di un paio di metodi per offrire la nostra configurazione specifica.
- **WebSecurityConfigurerAdapter**: questa classe offre una classe base per creare un Confiurazione di Spring
- **protected void configure(HttpSecurity http)**: QUesto metodo offre la sicurezza richiesta per differenti URL
- **antMatchers("/*secure*/**").access("hasRole('USER')")**: serve un ruolo da USER per accedere a qualsiasi URL che contenga la sottostringa "secure"
- **antMatchers("/login").permitAll()**: Permette a tutti gli utenti di accedere alla pagina di login
- **public void configureGlobalSecurity(AuthenticationManagerBuilder auth)**: in questo esempio stiamo usando un autenticatore in memoria. Può essere usato per connettersi ad un database (auth.jdbcAuthentication()) o un LDAP (auth.ldapAuthentication()), o un provider personalizzato
- **withUser("firstuser").password("password1")**: configura una combinazione User-PWD valido in memoria
- **.roles("USER","ADMIN")**: Assegna un ruolo ad un utente

Quando proviamo ad accedere ad'URL sicura, saremo reindirizzati alla pagina di login. Spring Security offre maniere di personalizzare la logica della pagina così come il reindirizzamento.

Aggiungere una funzionalità di logout

Spring Security offre soluzioni per permettere all'utente di fare il oog out ed essere reindirizzato verso pagine specifiche. L'URI del LogoutController solitamente è mappata sul link di logout della UI.

```
@Controller
public class LogoutController {

  @RequestMapping(value = "/secure/logout", method =
      ↪ RequestMethod.GET)
  public String logout(HttpServletRequest request,
    HttpServletResponse response) {

    Authentication auth =
        ↪ SecurityContextHolder.getContext()
      .getAuthentication();
    if (auth != null) {
      new SecurityContextLogoutHandler()
        .logout(request, response, auth);
      request.getSession().invalidate();
    }

    return "redirect:/secure/welcome";
  }
}
```

Notare:

- **if(auth != null)**: Controlla che l'autenticazione sia valida
- **new SecurityContextLogoutHandler().logout(request, response, auth)**: SecurityContextLogoutHandler esegue un logout rimuovendo le informazioni di autenticazione da SecurityContextHolder
- **return "redirect:/secure/welcome"**: reindirizza alla secure welcome page

2.3 Capitolo 5: Costruire dei microservizi con Spring Boot

2.3.1 Cos'è Spring boot?

- **Non** è un framework di generazione codice
- **Non** è un application/web server. Offre un'ottima integrazione con differenti tipi di applicazione e web server
- **Non** implementa nessun framework specifico

Quindi, per rispondere in maniera semplice a questa domanda, consideriamo qualche esempio.

Creare un veloce prototipo per un microservizio

Immaginiamo di voler creare un microservizio con Spring MVC e usare JPA (implementazione Hibernate) per connetterci ad un database. Consideriamo gli step:

1. Decidere la versione di Spring MVC, JPA e Hibernate
2. Preparare lo Spring Context per collegare tutti i livelli
3. Preparare il web layer con SPring MVC (DispatcherServlet, andler, resolvers, view resolvers, ecc.)
4. Preparare Hibernate nel data layer (SessionFactory, data source, ecc.)
5. Decidere e implementare come salvare la configurazione dell'applicazione, cosa varia tra vari ambienti
6. Decidere come eseguire lo unit testing
7. Decidere e implementare la strategy di gestione delle transazioni
8. Decidere e implementare la sicurezza

9. Preparare il framework di logging
10. Decidere e implementare come monitorare l'applicazione in produzione
11. Decidere e implementare le metriche di management per offrire statistiche riguardanti l'applicazione
12. Decidere e implementare come rilasciare l'applicazione su di un server web o applicazione

Tutte queste scelte richiedono anche settimane. SPring Boot vuole proprio risolvere questo problema offrendo una soluzione rapida per tutti i dettagli tecnici nello sviluppo dei microservizi.

Obiettivi primari

Gli obiettivi principali di Spring Boot sono:

- Permettere un inizio veloce
- Fare assunzioni basate sull'uso comune. Offre opzioni di conigurazione per gestire la deviazione dallo standard
- Offrire una pletora di feature non funzionali OOTB[4]
- Non usare la generazione di codice ed evitare di usare grandi configurazioni XML

Feature non funzionali

Alcune feature non funzionali offerte da Spring Boot sono:

- Gestione delle versioni e delle configurazioni di una pletora di framework, server e specifiche
- Opzioni di default per la sicurezza delle applicazioni

[4]Out Of The Box

- Metriche di default con possibilità di estensione
- Monitoraggio dell'applicazione usando health checks
- Multiple opzioni per una configurazione esterna

2.3.2 Spring Boot Hello World

I passi di base sono:

1. Configurare spring-boot-starter-parent nel file pom.xml.
2. Configurare il file pom.xml con gli starter di progetto richiesti
3. Configurare spring-boot-maven-plugin per permettere di lanciare l'applicazione
4. Creare la tua prima classe per avviare Spring Boot

Configurare spring-boot-starter-parent

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mastering.spring</groupId>
  <artifactId>springboot-example</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <name>First Spring Boot Example</name>

  <packaging>war</packaging>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
```

```
    <version>2.0.0.M1</version>
  </parent>

  <properties>
    <java.version>1.8</java.version>
  </properties>

  <repositories>
    <repository>
      <id>spring-milestones</id>
      <name>Spring Milestones</name>
      <url>https://repo.spring.io/milestone</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>

  <pluginRepositories>
    <pluginRepository>
      <id>spring-milestones</id>
      <name>Spring Milestones</name>
      <url>https://repo.spring.io/milestone</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>

</project>
```

Prima di tutto: perchè abbiamo bisogno di spring-boot-starter-parent? Perchè spring-boot-starter-parent contiene la versione base di Java in uso, la versione di default delle dipendenze che Spring Boot usa e la configurazione di base di Maven.

Configurare il file pom.xml con gli starter di progetto richiesti

Cos'è uno starter project? Sono dipendenze semplificate per differenti scopi. Per esempio, spring-boot-starter-web è lo starter per le applicazioni web, incluse le RESTful. Usa un container Tomcat di base.

Configurare spring-boot-maven-plugin

Quando costruiamo un'applicazione usando Spring Boot ci sono un paio di applicazioni possibli:

- Vogliamo lanciare l'applicazione sul nostro computer senza jar o war
- Vogliamo preparare dei file jar war da mandare in production successivamente

spring-boot-maven-plugin provvedere ad entrambe le situazioni. Possiamo configurarla molto velocemente:

```
<build>
  <plugins>
    <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Questo plugin provvede a svariati obiettivi di un'app, il più comune è, appunto, l'essere eseguita (mvn spring-boot:run nel prompt).

Creare la prima classe di avvio Spring Boot

```
package com.mastering.spring.springboot;
```

```
import org.springframework.boot.SpringApplication;
import
    ↪ org.springframework.boot.autoconfigure.SpringBootApplication;
import
    ↪ org.springframework.context.ApplicationContext;

@SpringBootApplication
public class Application {
  public static void main(String[] args){
    ApplicationContext ctx =
        ↪ SpringApplication.run(Application.class,
        ↪ args);
  }
}
```

Questa è la classe minima, semplicemente esegue il metodo run che prende in inflesso la classe nella quale è contenuto il main e l'array args in ingresso.

SpringApplication class La classe SpringApplication può essere usata per far partire un'applicazione Spring tramite un metodo main Java. I passaggi che sono tipicamente eseguiti sono:

1. Creare un'istanza dell'ApplicationContext
2. Abilitare le funzionalità per accettare da linea di comando argomenti ed esporli come proprietà di spring
3. Caricare tutti i bean configurati

L'annotazione @SpringBootApplication È una scorciatoia che accorpa tre annotazioni:

- **@Configuration**: Indica che è un file di configurazione Spring
- **@EnableAutoConfiguration**: Abilita l'auto configurazione, una feature importante per Spring Boot

- **@ComponentScan**: Abilita la ricerca dei bean nel package di questa classe e nei suoi subpackage

2.3.3 Starter projects

2.3.4 Cos'è REST?

Representional State Transfer (REST) è un insieme di linee guida per il web. Queste linee guida impongono dei limiti che assicurano al client un modo flessibile per interagire col server.

Alcuni di questi limiti sono:

- **Client-server**: Deve esserci sempre un sistema client-server per permettere una bassa dipendenza e evoluzione indipendente delle parti
- **Stateless**: Ogni servizio deve essere stateless
- **Interfaccia uniforme**: Ogni risorsa ha un identificatore. Nel caso dei servizi web è l'URI
- **Cache**: La risposta deve essere disponibile ad essere inserita nella cache. Ogni risposta deve dichiarare se può o no
- **Sistema a strati**: Il client non dovrebbe ottenere una connessione diretta al servizio. Poichè le richieste possono essere messe nella cache, il client potrebbe ottenere le risposte da un middle layer
- **Manipolazione delle risorse tramite la rappresentazione**: Una risorsa può avere multiple rappresentazioni. Dovrebbe essere possibile modificare le risorse attraverso messaggi con qualsiasi rappresentazione
- **HATEOAS**: Il client di un'applicazione RESTful dovrebbe conoscere solo una URL fissa. Tutte le risorse successive doverbbero essere raggiungibili tramite i link inclusi nella rappresentazione della risorsa

Starter	Descrizione
spring-boot-starter-web-services	Per sviluppare servizi web XML-based
spring-boot-starter-web	Per costruire applicazioni MVC-based o applicazioni Restful. Usa Tomcat come servlet
spring-boot-starter-activemq	Supporta comunicazioni basate sui mesasggi usando JMS su ActiveMQ
spring-boot-starter-integration	Supporta Spring Integration, usata per implementare enterprise integration pattern
spring-boot-starter-test	Offre supporto per svariati framework di testing, come Junit, mockito e Hamcrest
spring-boot-starter-jdbc	Ovvre supporto per Spring JDBC. Configura Tomcat JDBC pool come default
spring-boot-starter-validation	Offre supporto per le API per la validazione dei Java Bean. L'implementazione di default è quella di Hibernat
spring-boot-starter-hateoas	HATEOAS sta per Hypermedia As The Engine Of Application State. I servizi RESTful che usano HATEOA restituiscono link a risorse addizionali che sono correlate al contesto attuale in aggiunta ai dati
spring-boot-starter-jersey	JAX-RS è lo standard di Java Enterprise per sviluppare API REST. Jersey è l'implementazione di default
spring-boot-starter-websocket	HTTP è stateless. WebSocket permette di mantenere una connessione tra il server e il browser.
spring-boot-starter-aop	Offre supporto per la programmazione orientata agli asp Offre anche supporto per AspectJ per la programmazion orientata agli aspetti avanzata
spring-boot-starter-amqp	Con RabbitMQ come defaul, questo starter offre un sistema di messaggi basato su AMQP
spring-boot-starter-security	Offre la configurazione automatica di Spring Security
spring-boot-starter-data-jpa	Offre supporto per Spring Data JPA. L'implementazione di default è hibernate
spring-boot-starter	Supporto base per le applicazioni SpringBoot. Offre auto configurazione e logging
spring-boot-starter-batch	Offre supporto per sviluppare applicazioni batch usando Spring Batch
spring-boot-starter-cache	È il supporto base per il sistema di caching
spring-boot-starter-data-rest	Offre supporto per esporre servizi REST usando Spring Data REST

Una risposta di esempio è questa:

```
{
  "_embedded":{
    "todos":[{
      "user":"Jill",
      "desc":"Learn Hibernate",
      "done":false,
      "_links":{
        "self":{
          "href":"http://localhost:8080/todos/1"
          },
        "todo":{
          "href":"http://localhost:8080/todos/1"
        }
      }
    }
  ]
  },

  "_links":{
    "self":{
      "href":"http://localhost:8080/todos"
    },
    "profile":{
      "href":"http://localhost:8080/profile/todos"
    },
    "search":{
      "href":"http://localhost:8080/todos/search"
    }
  }
}
```


Questa risposta include i link a:

- To Do specifici: http://localhost:8080/todos/1
- Risorsa search: http://localhost:8080/todos/search

Se il client volesse fare una ricerca, avrebbe l'opzione di usare la URL per la ricerca dalla riposta e usarla.

2.3.5 Primo servizio REST

Non sarà perfettamente compliant alle norme indicate prima.

Iniziamo dalla base: una classe POJO con un campio chiamato "message" e un costruttore con un argomento:

```
package com.mastering.spring.springboot.bean;
public class WelcomeBean {

  private String message;

  public WelcomeBean(String message) {
    super();
    this.message = message;
  }

  public String getMessage() {
    return message;
  }
}
```

Ora creiamo un semplice Controller REST che ritorna una stringa:

```
@RestController
public class BasicController {

  @GetMapping("/welcome")
  public String welcome() {
    return "Hello World";
  }
}
```

Notiamo:

- **@RestController**: L'annotazione offre una combinazione di @ResponseBody e @Controller

Unit Testing

```
@RunWith(SpringRunner.class)
@WebMvcTest(BasicController.class)
public class BasicControllerTest {

  @Autowired
  private MockMvc mvc;

  @Test
  public void welcome() throws Exception {
    mvc
      .perform(MockMvcRequestBuilders.get("/welcome")
        .accept(MediaType.APPLICATION_JSON))
      .andExpect(status().isOk())
      .andExpect(content().string(equalTo("Hello
        ↪ World")));
  }
}
```

Quindi:

- **@RunWith(SpringRunner.class)**: è semplicemente una scorciatoia per SPringJUnit4ClassRunner. Lancia uno Spring Context
- **@WebMvcTest(BasicController.class)**: Può essere usata con SpringRunner per scrivere semplici test per i controller Spring MVC. Carica solo i bean annotati con le annotazioni legate a Spring MVC.
- **@Autowired private MockMvc mvc:**: Autowire sul MockMvc bean che può essere usato per creare richieste
- **mvc.perform(MockMvcRequestBuilders.get("/welcome") .accept(MediaType.APPLICATION_JSON))**: Esegue una richiesta a /welcome con il valore Accept dell'header application/json
- **andExpect(status().isOk())**: Si aspetta che lo status della risposta sia 200 (successo)

- **andExpect(content().string(equalTo("Hello World")))**: Aspetta che il contenuto della risposta sia "Hello World"

Integration testing

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = Application.class,
  webEnvironment =
     ↪ SpringBootTest.WebEnvironment.RANDOM_PORT)
public class BasicControllerIT {

  private static final String LOCAL_HOST =
    "http://localhost:";

  @LocalServerPort
  private int port;

  private TestRestTemplate template = new
     ↪ TestRestTemplate();

  @Test
  public void welcome() throws Exception {
    ResponseEntity<String> response = template
      .getForEntity(createURL("/welcome"),
         ↪ String.class);
    assertThat(response.getBody(), equalTo("Hello
       ↪ World"));
  }

  private String createURL(String uri) {
    return LOCAL_HOST + port + uri;
  }
}
```

Notiamo:

- **@SpringBootTest(classes = Application.class, webEnvironment = SpringBootTest.WebEnvironment .RANDOM_PORT)**: Offre funzionalità aggiuntive oltre al-

lo Spring TestContext. Offre supporto alla configurazione delle porte per il container

- **@LocalServerPort private int port:**
  SpringBootTest: Assicura che la porta sia autowired alla variabile porta
- **private String createURL(String uri)**: fa l'append dell'host locale e della porto all'URI in modo da creare una URL completa
- **private TestRestTemplate template**
  = new TestRestTemplate():
  TestRestTemplate: Usato tipicamente negli integration test. Offre funzionalità aggiuntive per il RestTemplate, che è specialmente utile nel contesto del integration test.
- **template.getForEntity(createURL("/welcome"),**
  String.class): Esegue una richiesta all'URI
- **assertThat(response.getBody(), equalTo("Hello World"))**: Si assicura che il response body sia "Hello World"

Semplice metodo REST che ritorna un oggetto

```
@GetMapping("/welcome-with-object")
public WelcomeBean welcomeWithObject() {
  return new WelcomeBean("Hello World");
}
```

Questo semplice metodo restituisce un WelcomeBean contenente il messaggio "Hello World".

Eseguire una richiesta Contattando http://localhost:8080/welcome-with-object otteniamo come risposta:

"message":"Hello World"

Ora la domanda è: com'è stato convertito automaticamente in un JSON? Fa parte dell'autoconfigurazione. Se Jackson è

nell classpath allora le istanze sono automaticamente tradotte in JSON e viceversa.

Unit Testing

```
@Test
public void welcomeWithObject() throws Exception {

  mvc.perform(
    MockMvcRequestBuilders.get("/welcome-with-object")
      .accept(MediaType.APPLICATION_JSON))
      .andExpect(status().isOk())
      .andExpect(content()
      .string(containsString("Hello World")));
}
```

Questo test p simile a quello di prima tranne per il fatto che sitamo usando "containsString" per controllare che effettivamente contenga "Hello WOrld". Ci sono modi migliori per scrivere test per i JSON.

Testing

```
@Test
public void welcomeWithObject() throws Exception {

  ResponseEntity<String> response =
    template.getForEntity(createURL("/welcome-with-object"),
      String.class);

  assertThat(response.getBody(),
    containsString("Hello World"));
}
```

Questo metodo è simile a quello di prima tranne per il fatto di fare un assert sulla sottostringa.

Metodo Get con variabili di percorso

Le variaibli di path sono usate per collegare un valore dall'URI ad una variabile del controller. In questo esempio vogliamo para-

metrizzare il nome in modo da poter personalizzare il messaggio di benvenuto:

```
private static final String helloWorldTemplate =
    ↪ "Hello World, %s!";

@GetMapping("/welcome-with-parameter/name/{name}")
public WelcomeBean welcomeWithParameter
  (@PathVariable String name){

  return new WelcomeBean
    (String.format(helloWorldTemplate, name));

}
```

Notiamo

- **@GetMapping("/welcome-with-parameter/name/name")**: name indicia il valore che sarà la variable. Possiamo avere più variabili nell'URI
- **welcomeWithParameter(@PathVariable String name)**: @PathVariable assicura che la variabile trovata nell'URI sia legata alla variabile name
- **String.format(helloWorldTemplate, name)**: Sostituisce il placeholder %s con il nome

Eseguire una richiesta Come prima, collegandoci a http://localhost:8080/welcome-with-parameter/name/**Buddy** otteniamo in risposta

```
{"message":"Hello World, Buddy!"}
```

Unit Testing

```
@Test
public void welcomeWithParameter() throws Exception {
  mvc.perform(
    MockMvcRequestBuilders
      .get("/welcome-with-parameter/name/Buddy")
      .accept(MediaType.APPLICATION_JSON))
```

```
      .andExpect(status().isOk())
      .andExpect(
        content().
        string(containsString("Hello World, Buddy")));
}
```

Notiamo:

- **MockMvcRequestBuilders.get ("/welcome-withparameter/name/Buddy")**: Fa una richiesta all'URI indicata
- **.andExpect(content() .string(containsString("Hello World, Buddy")))**: Controlla la risposta

Integration Testing

```
@Test
public void welcomeWithParameter() throws Exception {
  ResponseEntity<String> response = template
    .getForEntity(createURL("/welcome-with-parameter/name/Buddy"),String.class)

  assertThat(response.getBody(),
      ↪ containsString("Hello World, Buddy"));
}
```

Notiamo:

- **createURL("/welcome-with-parameter/name/Buddy")**: Crea una URL da usare successivamente
- **assertThat(response.getBody(), containsString("Hello World, Buddy"))**: Aspettiamo una risposta che contenga il messaggio col nome

2.3.6 Creare una risorsa todo

5

[5] It's the kind of voodoo—
That we few who do todo do!

HTTP request	Operazione
GET	**Read** – recupera i dettagli dalla risorsa
POST	**Create** – Crea un nuovo oggetto o risorsa
PUT	**Update**/Replace
PATCH	**Update**/modifica una parte di una risorsa
DELETE	**Delete**

Creeremo un servizio REST per una gestione base dei todo. I computi saranno:

- Recuperare la lista dei todo per utente
- Recuperare i dettagli per un todo specifico
- Creare un todo per l'utente

Metodi per le richieste, operazioni e URI

Una delle best practice per i servii REST è l'uso degli appropriati metodi HTTP basati sulle azioni che eseguiamo. Ricordiamo le richieste HTTP: Notiamo che corrispondono alle operazioni CRUD[6].

Bean e servizi

Per poter recuperare e salvare i dettagli di un todo dobbiamo usare un Todo bean e un servizio.

Prima il **bean** che accoglierà i dati:

```
public class Todo {

  private int id;
  private String user;
```

Is it evil?
Sure, it's true
Still, good things ensue
When you do todo todo todo!

[6]Create - Read - Update - Delete

```
  private String desc;
  private Date targetDate;
  private boolean isDone;

  public Todo() {}

  public Todo(int id, String user, String desc, Date
      ↪ targetDate, boolean isDone) {

    super();
    this.id = id;
    this.user = user;
    this.desc = desc;
    this.targetDate = targetDate;
    this.isDone = isDone;
  }

//ALL Getters
}
```

E ora invece il **servizio**:

```
@Service
public class TodoService {

  private static List<Todo> todos = new
      ↪ ArrayList<Todo>();
  private static int todoCount = 3;

  // Metodo avviato all'avvio
  static {
    todos.add(new Todo(1, "Jack", "Learn Spring MVC",
        ↪ new Date(), false));
    todos.add(new Todo(2, "Jack", "Learn Struts", new
        ↪ Date(), false));
    todos.add(new Todo(3, "Jill", "Learn Hibernate",
        ↪ new Date(), false));
  }

  // getAll
  public List<Todo> retrieveTodos(String user) {
```

```
    List<Todo> filteredTodos = new ArrayList<Todo>();

    for (Todo todo : todos) {
      if (todo.getUser().equals(user))
        filteredTodos.add(todo);
    }

    return filteredTodos;
  }

  // create
  public Todo addTodo(String name, String desc,
    Date targetDate, boolean isDone) {

      Todo todo = new Todo(++todoCount, name, desc,
          ↪ targetDate, isDone);
      todos.add(todo);

      return todo;
    }

  // get(id)
  public Todo retrieveTodo(int id) {
    for (Todo todo : todos) {
      if (todo.getId() == id)
        return todo;
    }

    return null;
  }
}
```

Il servizio non parla al database solo per comodità, teoricamente nei metodi GET ci sarebbe un'interrogazione.

Recuperare la lista dei todo

```
@RestController
public class TodoController {
```

```
  @Autowired
  private TodoService todoService;

  @GetMapping("/users/{name}/todos")
  public List<Todo> retrieveTodos
    (@PathVariable String name) {

    return todoService.retrieveTodos(name);
  }
}
```

Eseguire il servizio Interrogando http://localhost:8080/users/Jack/todos riceviamo:

```
[
  {"id":1,"user":"Jack","desc":"Learn Spring
      ↪ MVC","targetDate":1481607268779,"done":false},
  {"id":2,"user":"Jack","desc":"Learn
      ↪ Struts","targetDate":1481607268779,
      ↪ "done":false}
]
```

Unit testing

```
@RunWith(SpringRunner.class)
@WebMvcTest(TodoController.class)
public class TodoControllerTest {

  @Autowired
  private MockMvc mvc;

  @MockBean
  private TodoService service;

  @Test
  public void retrieveTodos() throws Exception {
    List<Todo> mockList = Arrays.asList(
      new Todo(1, "Jack", "Learn Spring MVC", new
          ↪ Date(), false),
```

```
      new Todo(2, "Jack", "Learn Struts", new Date(),
          ↪ false));

    when(service.retrieveTodos(anyString()))
      .thenReturn(mockList);

    MvcResult result = mvc
      .perform(MockMvcRequestBuilders
        .get("/users/Jack/todos")
        .accept(MediaType.APPLICATION_JSON))
      .andExpect(status().isOk())
      .andReturn();

    String expected = "["
      + "{id:1,user:Jack,desc:\"Learn Spring
          ↪ MVC\",done:false}" +","
      + "{id:2,user:Jack,desc:\"Learn
          ↪ Struts\",done:false}" + "]";

    JSONAssert.assertEquals(expected,
        ↪ result.getResponse().getContentAsString(),
        ↪ false);
  }
}
```

Notiamo:

- **Stiamo scrivendo un unit test**: Vogliamo testare solo le logiche del controller. Quindi inizializziamo un Mock MVC con solo il TodoController usando @WebMvcTest(TodoController.class)
- **@MockBean private TodoService service**: stiamo creando un mock del bean, in modo che non serva il servizio originale. Viene fatto tramite Mockito
- **when(service.retrieveTodos(anyString())) .thenReturn(mockList)**: Stiamo facendo il mock del metodo per restituire una lista al posto del return del metodo
- **MvcResult result = ...**: Stiamo trasormando il risultato della richiesta in un MvcResult in modo da poter fare asserzioni sulla risposta

- **JSONAssert.assertEquals(expected, result.getResponse().getContentAsString(), false)**: JSONAssert è un framework per eseguire assert sui JSON. Compara la risposta con il valore atteso

Integration testing

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = Application.class,
    ↪ webEnvironment =
    ↪ SpringBootTest.WebEnvironment.RANDOM_PORT)
public class TodoControllerIT {

  @LocalServerPort
  private int port;

  private TestRestTemplate template = new
      ↪ TestRestTemplate();

  @Test
  public void retrieveTodos() throws Exception {
    String expected = "[{id:1,user:Jack,desc:\"Learn
        ↪ Spring MVC\",done:false},
        ↪ {id:2,user:Jack,desc:\"Learn
        ↪ Struts\",done:false}" + "]";

    String uri = "/users/Jack/todos";

    ResponseEntity<String> response =
        ↪ template.getForEntity(createUrl(uri),
        ↪ String.class);

    JSONAssert.assertEquals(expected,
        ↪ response.getBody(), false);
  }

  private String createUrl(String uri) {
    return "http://localhost:" + port + uri;
  }
}
```

L'unica differenza dall'integration test precedente è il JSONAssert.

Recuperare i dettagli per un todo specifico

```
@GetMapping(path = "/users/{name}/todos/{id}")
public Todo retrieveTodo(@PathVariable String name,
    ↪ @PathVariable
  int id) {

  return todoService.retrieveTodo(id);
}
```

Notare che abbiamo due PathVariable.

Unit test

```
@Test
public void retrieveTodo() throws Exception {
  Todo mockTodo = new Todo(1, "Jack", "Learn Spring
     ↪ MVC", new Date(), false);

  when(service.retrieveTodo(anyInt()))
    .thenReturn(mockTodo);

  MvcResult result = mvc.perform(
    MockMvcRequestBuilders.get("/users/Jack/todos/1")
      .accept(MediaType.APPLICATION_JSON))
      .andExpect(status().isOk()).andReturn();

  String expected = "{id:1,user:Jack,desc:\"Learn
     ↪ Spring MVC\",done:false}";

  JSONAssert.assertEquals(expected,
     ↪ result.getResponse().getContentAsString()
     ↪ false);
}
```

Notiamo:

- **when(service.retrieveTodo(anyInt()))
.thenReturn(mockTodo)**: Creiamo un mock del servizio retriveTodo
- **MvcResult result = ...**: Accettiamo il risultato della request nella variabile per permetterci di eseguire assert
- **JSONAssert.assertEquals(expected,
result.getResponse()
.getContentAsString(), false)**: Assert sul risultato atteso

Integration testing

```
@Test
public void welcomeWithParameter() throws Exception {
  ResponseEntity<String> response =
      ↪ template.getForEntity(
  createURL("/welcome-with-parameter/name/Buddy"),
      ↪ String.class);

  assertThat(response.getBody(),
      ↪ containsString("Hello World, Buddy"));
}
```

Aggiungere un todo

Ricordiamo, per creare una risorsa si usa il metodo HTTP Post. Quindi:

```
@PostMapping("/users/{name}/todos")
ResponseEntity<?> add(@PathVariable String name,
  @RequestBody Todo todo) {

  Todo createdTodo = todoService
    .addTodo(name, todo.getDesc(),
        ↪ todo.getTargetDate(), todo.isDone());

  if (createdTodo == null) {
    return ResponseEntity.noContent().build();
  }
```

```
  URI location = ServletUriComponentsBuilder
    .fromCurrentRequest()
    .path("/{id}")
    .buildAndExpand(createdTodo.getId())
    .toUri();

  return ResponseEntity.created(location).build();
}
```

Notiamo:

- **@PostMapping("/users/name/todos")**: @PostMapping è usato per mappare direttamente le richieste POST
- **ResponseEntity<?> add(@PathVariable String name, @RequestBody Todo todo)**: Una richiesta HTTP POST dovrebbe idealmente ritornare l'URI della risorsa creata. Usiamo ResourceEntity per fare ciò. @RequestBody collega il body della richiesta direttamente al bean
- **ResponseEntity.noContent().build()**: Usato per ritornare che la creazione della risorsa è fallita
- **ServletUriComponentsBuilder**
  .fromCurrentRequest()
  .path("/id")
  .buildAndExpand(createdTodo.getId())
  .toUri(): Compone l'URI della risorsa creata che può essere restituito dalla risposta
- **ResponseEntity.created(location).build()**: Ritorna lo status 201 (CREATED) con il link della risorsa

Postman Postman è un servizio che permette di testare le API.

Per creare un todo usando POST, dovremmo includere un JSON per il todo nel body della richiesta. L'interfaccia è molto intuitiva: si sceglie il tipo di richiesta sopra a sinistra, si sceglie formato raw e si avvia.

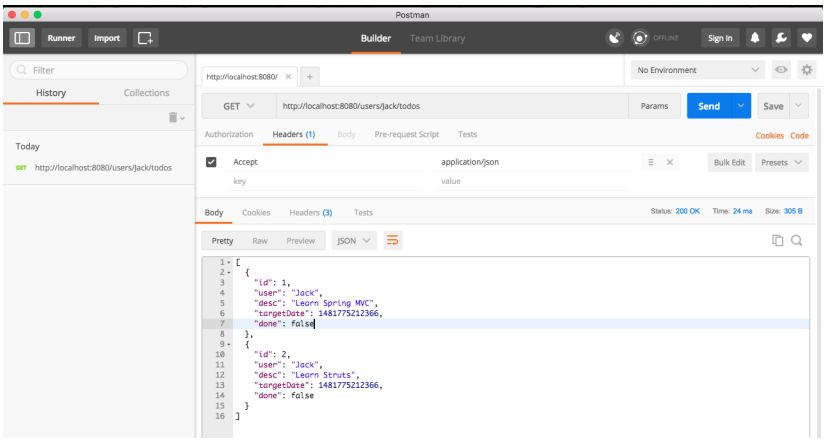

Figura 2.9: Postman

Unit testing

```
@Test
public void createTodo() throws Exception {

  Todo mockTodo = new Todo(CREATED_TODO_ID, "Jack",
      ↪ "Learn Spring MVC", new Date(), false);

  String todo = "{"user":"Jack","desc":"Learn Spring
      ↪ MVC", "done":false}";

  when(service.addTodo(anyString(), anyString(),
      ↪ isNull(),anyBoolean()))
    .thenReturn(mockTodo);

  mvc
  .perform(MockMvcRequestBuilders
    .post("/users/Jack/todos")
    .content(todo)
    .contentType(MediaType.APPLICATION_JSON)
  )
    .andExpect(status().isCreated())
    .andExpect(
      header()
```

```
        .string("location",
            ↪ containsString("/users/Jack/todos/" +
            ↪ CREATED_TODO_ID)));
}
```

Notiamo:

- **String todo = ""user":"Jack","desc":"Learn Spring MVC","done":false"**: Prepara il contenuto da postare
- **when(service .addTodo(anyString(), anyString(), isNull(), anyBoolean())) .thenReturn(mockTodo)**: Crea un mock del servizio per ritornare un todo dummy
- **MockMvcRequestBuilders.post("/users/Jack/todos") .content(todo) .contentType(MediaType.APPLICATION_JSON))**: Crea un POST riferito ad una URI specifica con il content type stabilito
- **andExpect(status().isCreated())**: Attende che lo status sia creato
- **andExpect(header().string("location", containsString("/users/Jack/todos/" + CREATED_TODO_ID)))**: Attende che l'hader contenga location con l'URI della risorsa creata

Integration testing

```
@Test
public void addTodo() throws Exception {

  Todo todo = new Todo(-1, "Jill", "Learn
     ↪ Hibernate", new Date(), false);

  URI location = template
    .postForLocation(createUrl("/users/Jill/todos")
       ↪ ,todo);

  assertThat(location.getPath(),
     ↪ containsString("/users/Jill/todos/4"));
}
```

- **URI location = template.postForLocation(createUrl("/users/Jill/todos"), todo)**: postForLocation è un metodo di utilità utile nei test per creare nuove risorse. Stiamo postando il todo verso l'URI per ricevere la location dall'header
- **assertThat(location.getPath(), containsString("/users/Jill/todos/4"))**: fa un assert sulla location contenente il path della nuova risorsa

2.4 Capitolo 6: Estendere i microservizi

2.4.1 Gestione delle eccezioni

Gestione delle eccezioni di default di Spring Boot

Risorsa non esistente Immaginiamo di chiamare una URL non esistente (http://localhost:8080/non-existing-resource), avremmo come risposta:

```
{
  "timestamp": 1484027734491,
  "status": 404,
  "error": "Not Found",
  "message": "No message available",
  "path": "/non-existing-resource"
}
```

Notiamo lo status 404.

Eccezione lanciata da una risorsa Creiamo una risorsa che lancia un'eccezione e inviamo una richiesta GET per capire come l'applicazione reagisca alle eccezioni a runtime.

```
@GetMapping(path = "/users/dummy-service")
public Todo errorService() {
  throw new RuntimeException("Some Exception
      ↪ Occured");
}
```

Notiamo:

- Stiamo creando un servizio GET con l'URI /users/dummy-service
- Il servizio lancia una RuntimeException

Se ora contattiamo http://localhost:8080/users/dummy-service la risposta sarà:

```
{
  "timestamp": 1484028119553,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "java.lang.RuntimeException",
  "message": "Some Exception Occured",
  "path": "/users/dummy-service"
}
```

Notiamo lo status 500 e, soprattutto, il messaggio dell'eccezione.

Lanciare un'eccezione personalizzata

```
public class TodoNotFoundException extends
    ↪ RuntimeException {
  public TodoNotFoundException(String msg) {
    super(msg);
  }
}
```

Questa è un'eccezione personalizzata base.

Ora potenziamo il nostro controller in modo che lanci l'eccezione quando un todo con l'ID passato non esiste:

```
@GetMapping(path = "/users/{name}/todos/{id}")
public Todo retrieveTodo(@PathVariable String name,
  @PathVariable int id) {
  Todo todo = todoService.retrieveTodo(id);
  if (todo == null) {
    throw new TodoNotFoundException("Todo Not Found");
```

```
  }

  return todo;
}
```

Quindi ora, nel caso il todo non venga trovato, verrà lanciata un'eccezione personalizzata.

Personalizzare il messaggio di errore

```
public class ExceptionResponse extends
    ↪ RuntimeException{

  private Date timestamp = new Date();
  private String message;
  private String details;

  public ExceptionResponse(String message, String
      ↪ details) {
    super();
    this.message = message;
    this.details = details;
  }

  public Date getTimestamp() {
    return timestamp;
  }

  public String getMessage() {
    return message;
  }

  public String getDetails() {
    return details;
  }
}
```

Così l'eccezione viene autonomamente popolata.

Quando la TodoNotFoundException viene lanciata vogliamo che venga restituita la risposta usando il bean ExceptionRespon-

se. Possiamo creare una gestione globale delle eccezioni tramite un ControllerAdvice:

```
@ControllerAdvice
@RestController
public class RestResponseEntityExceptionHandler
    ↪ extends ResponseEntityExceptionHandler{

  @ExceptionHandler(TodoNotFoundException.class)
  public final ResponseEntity<ExceptionResponse>
      ↪ todoNotFound(TodoNotFoundException ex) {

    ExceptionResponse exceptionResponse = new
        ↪ ExceptionResponse( ex.getMessage(), "Any
        ↪ details you would want to add");

  return new ResponseEntity<ExceptionResponse>
      ↪ (exceptionResponse, new HttpHeaders()
      ↪ HttpStatus.NOT_FOUND);
  }
}
```

Notiamo:

- **RestResponseEntityExceptionHandler extends ResponseEntityExceptionHandler**: Estendiamo ResponseEntityExceptionHandler, che è la classe base offerta da Sring MVC per centralizzare la gestione delle eccezioni nei ControllerAdvice
- **@ExceptionHandler(TodoNotFoundException.class)**: Definisce che il metodo che segue gestirà tutte le eccezioni di TodoNotFoundException.class
- **ExceptionResponse exceptionResponse = new ExceptionResponse(ex.getMessage(),"Any details you would want to add")**: Crea una nuova exception response
- **new ResponseEntity<ExceptionResponse> (exceptionResponse,new HttpHeaders(), HttpStatus.NOT_FOUND)**: Restituisce una risposta 404 resource not found

Situazione	Response status
La request body non incontra le specifiche dell'API, Non contiene dettagli sufficienti o contiene errori di validazione	400 BAD REQUEST
Autenticazione o autorizzazione fallita	401 UNAUTHORIZED
L'tente non può eseguire una data operazione a causa di vari fattori come una richiesta eccessiva per i suoi limiti	402 FORBIDDEN
La risorsa non esiste	404 NOT FOUND
Operazione non supportata, per esempio, provare un POST su di una risorsa dove solo GET è consentito	405 METHOD NOT ALLOWED
Errore del server. Idealmente non dovrebbe accadere. L'utente non dovrebbe poter fare niente per risolvere questo problema	500 INTERNAL SERVER ERROR
:D	418 I'M A TEAPOT

Ora quando contatteremo http://localhost:8080/users/Jack/todos/222 riceveremo come risposta:

```
{
  "timestamp": 1484030343311,
  "message": "Todo Not Found",
  "details": "Any details you would want to add"
}
```

Se volessimo gestire tutte le eccezioni in un modo specifico avremmo:

```
@ExceptionHandler(Exception.class)
public final ResponseEntity<ExceptionResponse>
    ↪ todoNotFound(
  Exception ex) {
  //Customize and return the response
}
```

Ricordiamo una cosa: specifico > generale, quindi se ci sarà una gestione specifica partirà quella, altrimenti questa per tutte le altre.

2.4.2 HATEOAS

HATEOAS è uno degli obblighi delle applicazioni REST.

Spiegato in parole semplice, HATEOAS impone che le API siano navigabili come se fossero una pagina web, devono avere quindi la proprietà di "discoverability", ovvero si deve poter conoscere l'insieme di azioni praticabili.

Ad esempio, questa è la risposta ricevuta visitando http://localhost:8080/todos:

```
{
  "_embedded" : {
    "todos" : [ {
      "user" : "Jill",
      "desc" : "Learn Hibernate",
      "done" : false,
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/todos/1"
        },
        "todo" : {
          "href" : "http://localhost:8080/todos/1"
        }
      }
    }]
  } ,

  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/todos"
    },
    "profile" : {
      "href" : "http://localhost:8080/profile/todos"
    },
    "search" : {
      "href" : "http://localhost:8080/todos/search"
    }
  },
}
```

Se l'utente vuole fare una ricerca ha l'opzione per prendere l'URL di ricerca dalla response e inviare una richiesta ad essa. Ciò ridurrebbe il coupling tra i provider e il consumer.

Inviare HATEOAS link in risposta

Per fare un esempio su come costruirlo, potenziamo retrieveTodo (/users/name/todos/id) in modo che restituisca anche un link per recuperare tutti i todo (/users/name/todos):

```
@GetMapping(path = "/users/{name}/todos/{id}")
public Resource<Todo> retrieveTodo(
  @PathVariable String name,
  @PathVariable int id) {

  Todo todo = todoService.retrieveTodo(id);
  if (todo == null) {
    throw new TodoNotFoundException("Todo Not Found");
  }

  Resource<Todo> todoResource = new
     ↪ Resource<Todo>(todo);

  ControllerLinkBuilder linkTo =
     ↪ linkTo(methodOn(this.getClass()).retrieveTodos(name));

  todoResource.add(linkTo.withRel("parent"));

  return todoResource;
}
```

Notiamo:

- **ControllerLinkBuilder linkTo = linkTo(methodOn(this.getClass()) .retrieveTodos(name))**: Vogliamo ottenere il link collegato al metodo retrieveTodos
- **linkTo.withRel("parent")**: La relazione con la corrente relazione è quella di "parent"

Ora la risposta a hettp://localhost:8080/users/Jac/todos/1 sarà:

```
{
  "id": 1,
  "user": "Jack",
  "desc": "Learn Spring MVC",
  "targetDate": 1484038262110,
  "done": false,

  "_links": {
    "parent": {
      "href": "http://localhost:8080/users/Jack/todos"
    }
  }
}
```

2.4.3 Validazione

Ogni buon servizio validerà sempre i dati prima di processarli (SQL injection, anyone?).

Le Bean Validation API offrono un numero di annotazioni che possono essere usate per validare i bean.

Abilitare la validazione su di un metodo di un controller

```
@RequestMapping(method = RequestMethod.POST, path =
    ↪ "/users/{name}/todos")
ResponseEntity<?> add(@PathVariable String name
  @Valid @RequestBody Todo todo) {

  ...

}
```

L'annotazione @Valid è usata per segnare un parametro per la validazione. Ogni validazione è definita nel bean Todo è eseguita prima che ilmetodo add sia eseguito.

Definire la validazione del bean

Aggiungiamo le validazioni sui campi del Todo bean:

```
public class Todo {

  private int id;

  @NotNull
  private String user;

  @Size(min = 9, message = "Enter at least 10
     ↪ Characters.")
  private String desc;
```

Notiamo:

- **@NotNull**: Controlla che il campo non sia vuoto
- **@Size(min = 9,**
  message = "Enter at least 10 Characters."): Controlla che il campo abbia almeno 10 caratteri

Annotazioni per la validazione

Ci sono svariate annotazioni, eccone alcune:

- **@AssertFalse, @AssertTrue**: Per i booleani. Controlla il loro valore
- **@Assert**: controlla che il valore sia vero
- **@Future**: La data deve essere nel futuro
- **@Past**: Controlla che la data sia nel passato
- **@Max**: annota un numero che deve essere minore o uguale al valore specificato
- **@Min**: annota un numero che deve essere maggiore o uguale al valore specificato

- **@NotNull**: Il valore non può essere null
- **@Pattern**: La stringa annotata deve rispettare la regex collegata
- **@Size**: L'elemento deve rientrare in una lunghezza massima

Unit Testing

```
@Test
public void createTodo_withValidationError() throws
    ↪ Exception {

  Todo mockTodo = new Todo(CREATED_TODO_ID, "Jack",
      ↪ "Learn Spring MVC", new Date(), false);

  String todo =
      ↪ "{"user":"Jack","desc":"Learn","done":false}";

  when( service.addTodo(anyString(), anyString(),
      ↪ isNull(), anyBoolean()))
    .thenReturn(mockTodo);

  MvcResult result =
      ↪ mvc.perform(MockMvcRequestBuilders
    .post("/users/Jack/todos")
      .content(todo)
      .contentType(MediaType.APPLICATION_JSON))
    .andExpect(status().is4xxClientError())
    .andReturn();
}
```

Notiamo:

- **"desc":"Learn"**: Stiamo usando una descrizione con lunghezza 5 quando abbiamo detto che deve essere almeno 10 caratteri
- **.andExpect(status().is4xxClientError())**: Controlla per gli errori di validazione

2.4.4 Documentare un servizio REST

Un "contratto" è ciò che definisce i dettagli del servizio:

- Come posso chiamarlo? Qual è l'URI?
- Quale dovrebbe essere il formato della richiesta?
- Che tipo di risposta dovrei attendere?

Ci sono svariate opzioni per definire un contratto per un servizio RESTful. Il più popolare è Swagger.

Creare una specifica di Swagger

Springfox Swagger può essere utilizzato per generare la documentazione dal codice. In più c'è uno strumento chiamato Swagger UI che, integrato con l'applicazione, offre documentazione leggibile.

Semplicemente bisogna inserire nel pom.xml:

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.4.0</version>
</dependency>

<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.4.0</version>
</dependency>
```

Lo step successivo è quello di abilitare e generare la documentazione swagger:

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {
```

```
  @Bean
  public Docket api() {
    return new Docket(DocumentationType.SWAGGER_2)
      .select()
      .apis(RequestHandlerSelectors.any())
      .paths(PathSelectors.any()).build();
  }
}
```

Notiamo:

- **@Configuration**: Crea un file di coniguazione Spring
- **@EnableSwagger2**: Abilita swagger
- **Docket**: Un semplice builder per configurare la generazione della documentazione swagger usando Swagger Spring MVC Framework
- **new Docket(DocumentationType.SWAGGER_2)**: Configura Swagger 2 come versione
- **.apis(RequestHandlerSelectors.any())** **.paths(PathSelectors.any())**: Include tutte le API e i path nella documentazione

Quindi, quando interrogheremo http://localhost:8080/v2/api-docs, otteremo: Ad esempio, questa è la documentazione per recupeare il servizio todos:

```
"/users/{name}/todos": {
  "get": {
    "tags": [
      "todo-controller"
    ],

    "summary": "retrieveTodos",
    "operationId": "retrieveTodosUsingGET",

    "consumes": [
```

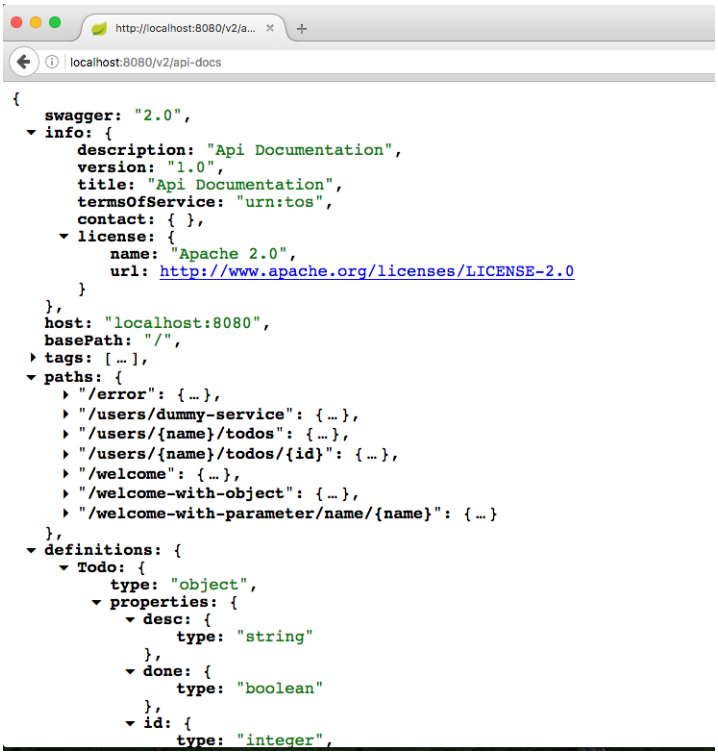

Figura 2.10: Documentazione Swagger

```
    "application/json"
  ],

  "produces": [
    "*/*"
  ],

  "parameters": [{
    "name": "name",
    "in": "path",
    "description": "name",
    "required": true,
    "type": "string"
  }],

"responses": {
```

```
    "200": {
      "description": "OK",
      "schema": {
        "type": "array",
        items": {
          "$ref": "#/definitions/Todo"
        }
      }
    },

    "401": {
      "description": "Unauthorized"
    },

    "403": {
      "description": "Forbidden"
    },

    "404": {
      "description": "Not Found"
    }
  }
}
```

Il servizio definisce una richiesta e una risposta al servizio. Sono anche definite le differenti risposte ai vari status.

Questa invece la definizione del bean Todo:

```
"Resource<Todo>": {
  "type": "object",
  "properties": {
    "desc": {
      "type": "string"
    },

    "done": {
      "type": "boolean"
    },

    "id": {
```

```
      "type": "integer",
      "format": "int32"
    },

    "links": {
      "type": "array",
      "items": {
        "$ref": "#/definitions/Link"
      }
    },

    "targetDate": {
      "type": "string",
      "format": "date-time"
    },

    "user": {
      "type": "string"
    }
  }
}
```

Swagger UI Swagger UI è un modo comodissimo per avere una vista molto intuitiva dei vari servizi e degli endpoint. Offre la possibilità di visualizzare la documentazione (Swagger JSON) in maniera alternativa.

Notiamo:

- **Parameters** mostra tutti i parametri importanti inclusi quelli del request body
- **Parameters type** mostra tutta la struttura del body della richiesta
- **Response messages** mostra differenti status HTTP ritornati dal servizio

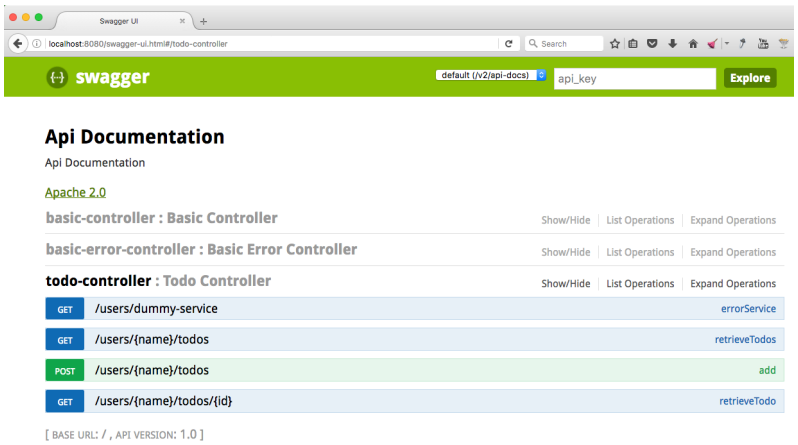

Figura 2.11: Visualizzazione della documentazione

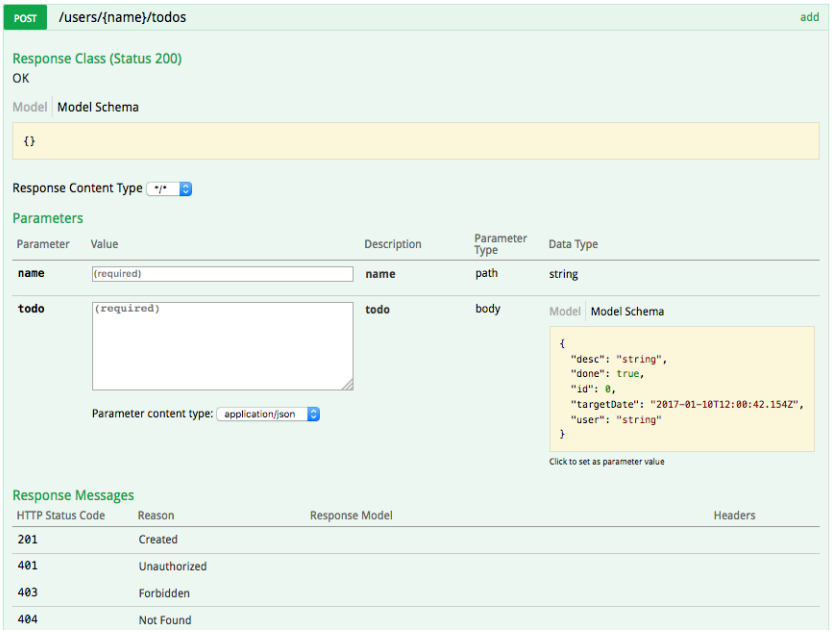

Figura 2.12: Esempio di elemento POST

Personalizzare la documentazione con le annotazioni Swagger offre delle annotazioni che vengono aggiunte ai servisi RESTful per personalizzare la documentazione:

```
@ApiOperation(
  value = "Retrieve all todos for a user by passing
     ↪ in his name",
  notes = "A list of matching todos is returned.
     ↪ Current pagination is not supported."
     ↪ response = Todo.class, responseContainer =
     ↪ "List" produces = "application/json")
@GetMapping("/users/{name}/todos")
public List<Todo> retrieveTodos(@PathVariable String
    ↪ name) {
  return todoService.retrieveTodos(name);
}
```

Notiamo:

- **@ApiOperation(value = "Retrieve all todos for a user by passing in his name")**: Produce la documentazione come un riassunto del servizio
- **notes = "A list of matching todos is returned. Current pagination is not supported."**: Prodotto nella documentazione come descrizione del servizio
- **produces = "application/json"**: Personalizza la sezione produces della documentazione

```
# Documentazione prima

"get": {
  "tags": [
  "todo-controller"
  ],
  "summary": "retrieveTodos",
  "operationId": "retrieveTodosUsingGET",
  "consumes": [
    "application/json"
  ],
  "produces": [
    "*/*"
  ]
```

```
  ...
}

# Documentazione dopo
get": {
  "tags": [
    "todo-controller"
  ],
  "summary": "Retrieve all todos for a user by
      ↪ passing in his name",
  "description": "A list of matching todos is
      ↪ returned. Current pagination is not
      ↪ supported.",
  "operationId": "retrieveTodosUsingGET",
  "consumes": [
    "application/json"
  ],
  "produces": [
    "application/json",
  "*/*"
  ]

  ...
}
```

Lista di annotazioni Swagger

- **@Api**: Segna la classe come risorsa per Swagger
- **@ApiModel**: Offre informazioni addizionali sul modello
- **@ApiModelProperty**: Agiunge e manipola i dati delle proprietà di un model
- **@ApiOperation**: Descrive un'operazione o un metodo HTTP rispetto allo specifico path
- **@ApiParam**: Agginge dei metadati addizionali per i parametri dell'operazione

- **@ApiResponse**: Decrive un esempio di risposta di un operazione
- **@ApiResponses**: Un wrapper per consentire una lista di più oggetti @ApiResponse
- **@Authorization**: Dichiara uno schema di autorizzazione da utilizzare su di una risorsa o un'operazione
- **@AuthorizationScope**: Descrive uno scope di autorizzazione OAuth 2
- **@ResponseHeader**: Rappresenta l'header che può essere fornito come parte della risposta

In più Swagger offre una serie di annotazioni che possono essere usate per personalizzare le informazioni di alto livello riguardanti un gruppo di servizi (contatti, licenze e altre informazioni). Tra questi:

- **@SwaggerDefinition**: proprietà a livello di definizione che può essere aggiunta per generare una definizione di Swagger
- **@Info**: Metadati generali per una definizione di Swagger
- **@Contact**: Proprietà per descrivere una persona che deve essere contattata per una definizione di Swagger
- **@License**: Proprietà per descrivere una licenza per una proprietà di Swagger

2.4.5 Mettere in sicurezza un servizio REST con Spring Security

Da inserire tramite file pom.xml:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Autenticazione base

Spring security blocca tutti i servizi in maniera automatica. La risposta base ad ogni richiesta è:

```
{
  "timestamp": 1484120815039,
  "status": 401,
  "error": "Unauthorized",
  "message": "Full authentication is required to
     ↪ access this
  resource",
  "path": "/users/Jack/todos"
}
```

Spring richiede quindi uno user ID e una password. Non avendole ancora impostate, Spring security imposta automaticamente uno user la cui password è nel log di sistema.

Con postman possiamo inviare una richiesta con i dati di autenticazione.

Possiamo anche configurare lo userID e la password in application.properties. Spring security offre anche la possibilità di autenticarsi tramite LDAP o JDBC.

Integration testing

```
private TestRestTemplate template = new
    ↪ TestRestTemplate();
HttpHeaders headers = createHeaders("user-name",
    ↪ "user-password");

HttpHeaders createHeaders(String username, String
    ↪ password) {
  return new HttpHeaders() {{
    String auth = username + ":" + password;
    byte[] encodedAuth =
        ↪ Base64.getEncoder().encode(auth
      .getBytes(Charset.forName("US-ASCII")));
    String authHeader = "Basic " + new
        ↪ String(encodedAuth);
```

```
    set("Authorization", authHeader);
    }
  };
}

@Test
public void retrieveTodos() throws Exception {
  String expected = "["
    + "{id:1,user:Jack,desc:\"Learn Spring
       ↪ MVC\",done:false}" + ","
    + "{id:2,user:Jack,desc:\"Learn
       ↪ Struts\",done:false}" + "]";

  ResponseEntity<String> response =
      ↪ template.exchange(
    createUrl("/users/Jack/todos"), HttpMethod.GET,
       ↪ new HttpEntity<String>(null, headers),
       ↪ String.class);

  JSONAssert.assertEquals(expected,
      ↪ response.getBody(), false);
}
```

Notiamo:

- **createHeaders("user-name", "user-password")**: crea un header di autenticazione in base64[7]
- **ResponseEntity<String> response = template.exchange(createUrl("/users/Jack/todos"), HttpMethod.GET,new HttpEntity<String>(null, headers), String.class)**: HttpEntity offre l'header che abbiamo creato prima al template REST

Unit testing

```
@RunWith(SpringRunner.class)
@WebMvcTest(value = TodoController.class, secure =
    ↪ false)
```

[7]Base64 è un sistema di codifica che consente la traduzione di dati binari in stringhe di testo ASCII, rappresentando i dati sulla base di 64 caratteri ASCII diversi. Viene usato principalmente come codifica di dati binari nelle e-mail, per convertire i dati nel formato ASCII.

```
public class TodoControllerTest {
```

Semplicemente mettere "secure = false" disabilità spring secure per il test.

Autenticazione Oauth 2

Permette ad applicazioni terze di ottenere informazioni ristrette dell'utente tramite il servizio.

Consideriamo un esempio. Vogliamo esporre le nostre API su internet. In questo scambio OAuth 2 sono importanti queste parti:

- **Resource owner**: È l'utente dell'applicazione di terze parti che vuole usare le nostre API. Decide quante informazioni rendere disponibili alle API
- **Resource server**: Offre le API, le risorse che vogliamo mettere al sicuro
- **Client**: L'applicazione dei terze parti che vuole usare le API
- **Authorization server**: È il server che offre il servizio OAuth

Flow di alto livello

Il flow è:

1. L'applicazione richiede che l'utente autorizzi l'accesso alle API
2. L'utente offre l'accesso, l'applicazione riceve il consenso
3. L'applicazione offre all'utente il consenso all'autorizzazione e le sue credenziali all'authorization server
4. Se l'autenticazione è positiva, l'authorization server risponde con un token d'accesso

5. L'applicazione chiama le API (il resource server) che offre il token d'accesso per l'autenticazione

6. Se il token è valido, le risorse ritornano i dettagli della risorsa

Implementare l'autenticazione OAuth 2

Solitamente il server di autorizzazione sarebbe differente rispetto a quello dove l'API è esposta.

Questo snippet mostra come abilitare l'applicazione ad agire come server di autenticazione:

```
@EnableResourceServer
@EnableAuthorizationServer
@SpringBootApplication
public class Application {
```

Notiamo:

- **@EnableResourceServer**: Abilita un filtro di Spring Security che autentica le richieste tramite un token OAuth 2

- **@EnableAuthorizationServer**: Abilita un EuthorizationEndpoint e un TokenEndpoint nel contesto dell'applicazione corrente, che sarà il DispatcherServlet

Ora possiamo configurare i dettagli dell'accesso in application.properties:

```
security.user.name=user-name
security.user.password=user-password
security.oauth2.client.clientId: clientId
security.oauth2.client.clientSecret: clientSecret
security.oauth2.client.authorized-grant-types:
authorization_code,refresh_token,password
security.oauth2.client.scope: openid
```

Notiamo:

- **security.user.name and security.user.password**: sono i dettagli per il resource owner
- **security.oauth2.client.clientId and security.oauth2.client.clientSecret**: sono i dettagli per l'autenticazione del client che è definito nelle applicazioni di terze parti

Eseguire una richiesta OAuth Abbiamo bisogno di due passaggi:

1. Ottenere un token di accesso
2. Eseguire una richiesta usando il token di accesso

Ottenere un token di accesso Per ottenere il token chiamiamo il server passando i dettagli di autenticazione. Inviando in POST i dettagli della richiesta otteniamo come risposta qualcosa di simile a ciò:

```
{
  "access_token":
     ↪ "a633dd55-102f-4f53-bcbd-a857df54b821",
  "token_type": "bearer",
  "refresh_token":
     ↪ "d68d89ec-0a13-4224-a29b-e9056768c7f0",
  "expires_in": 43199,
  "scope": "openid"
}
```

Notiamo:

- **access_token**: Le applicazioni client possono usare il token di accesos per autenticare altre chiamate API. Solitamente il token scade in poco

- **refresh_token**: Le applicazioni client possono passare una nuova richiesta al server di autenticazione con il refresh_token per ottenere un nuovo access_token.

Integration test

```
@Test
public void retrieveTodos() throws Exception {
  String expected = "["
    + "{id:1,user:Jack,desc:\"Learn Spring
       ↪ MVC\",done:false}" + ","
    +"{id:2,user:Jack,desc:\"Learn
       ↪ Struts\",done:false}" + "]";

  String uri = "/users/Jack/todos";
  ResourceOwnerPasswordResourceDetails resource =
    new ResourceOwnerPasswordResourceDetails();

  resource.setUsername("user-name");
  resource.setPassword("user-password");
  resource.setAccessTokenUri(createUrl("/oauth/token"));
  resource.setClientId("clientId");
  resource.setClientSecret("clientSecret");
  resource.setGrantType("password");

  OAuth2RestTemplate oauthTemplate = new
     ↪ OAuth2RestTemplate(resource,new
     ↪ DefaultOAuth2ClientContext());

  ResponseEntity<String> response =
     ↪ oauthTemplate.getForEntity(createUrl(uri),
     ↪ String.class);

  JSONAssert.assertEquals(expected,
     ↪ response.getBody(), false);
}
```

Notiamo:

- **ResourceOwnerPasswordResourceDetails resource = new ResourceOwnerPasswordResourceDetails()**: Prepa-

riamo ResourceOwnerPasswordResourceDetails con le credenziali dell'utente e del client

- **resource.setAccessTokenUri(createUrl("/oauth/token"))**: Configura l'URL del server di autenticazione
- **OAuth2RestTemplate oauthTemplate = new OAuth2RestTemplate(resource, new DefaultOAuth2ClientContext())**: OAuth2RestTemplate è un'estensione del RestTemplate, che supporta il protocollo OAuth 2

2.4.6 Internazionalizzazione

È il processo di sviluppare un'applicazione e i servizi in modo che siano personalizzati per lingue differenti e culture nel mondo. È anche chiamata localizzazione. Spring Boot ha un servizio di internazionalizzazione OOTB.

Abbiamo prima di tutto bisogno di un LocaleResolver:

```
@Bean
public LocaleResolver localeResolver() {
  SessionLocaleResolver sessionLocaleResolver = new
     ↪ SessionLocaleResolver();
  sessionLocaleResolver.setDefaultLocale(Locale.US);

  return sessionLocaleResolver;
}

@Bean
public ResourceBundleMessageSource messageSource() {
  ResourceBundleMessageSource messageSource = new
     ↪ ResourceBundleMessageSource();
  messageSource.setBasenames("messages");
  messageSource.setUseCodeAsDefaultMessage(true);

  return messageSource;
}
```

Notiamo:

- **sessionLocaleResolver.setDefaultLocale(Locale.US)**: Settiamo il default locale con Locale.US
- **messageSource.setBasenames("messages")**: Settiamo il nome della fonte di base dei messaggi come messages. Le varie fonti vengono prese dai file message_[sigla locale].properties.
- **messageSource.setUseCodeAsDefaultMessage(true)**: Se un messagio non è trovato allora il codice è ritornato come messaggi di default

I vari file semplicemente conterranno i campi con la loro definizione, ad esempio:

```
# message_en.properties

welcome.message=Welcome in English

# message_fr.properties

welcome.message=Welcome in French
```

Per poi invocarlo in maniera semplice con il locale specificato nell'header "Accept-Language":

```
@GetMapping("/welcome-internationalized")
public String msg(@RequestHeader(value =
    ↪ "Accept-Language", required = false) Locale
    ↪ locale) {

  return messageSource.getMessage("welcome.message",
      ↪ null, locale);
}
```

Notiamo:

- **@RequestHeader(value = "Accept-Language", required = false) Locale locale**: Il locale è confiurato tramite la request header Accept-Language. Non è essenziale. Se un locale non è specificato viene usato quello di default.

- **messageSource.getMessage("welcome.message", null, locale)**: messageSource è autowired nel controller.

2.4.7 Caching

La cache ha un ruolo essenziale nelle performance e nalla scalabilità delle applicazioni. Spring offre un'astrazione basata sulle annotazioni.

Spring-boot-starter-cache

Come sempre basta una piccola agigunta a pom.xml:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

Abilitare il caching

```
@EnableCaching
@SpringBootApplication
public class Application {
```

Yup, basta @EnableCaching.

Caching data

Ora che abbiamo abilitato il caching possiamo aggiungere l'annotazione @Cachable ai metodi di cui vogliamo mettere in cache i dati.

```
@Cacheable("todos")
public List<Todo> retrieveTodos(String user) {
```

I Todo per uno specifico utente sono messi nella cache. Sono anche ammesse condizioni:

```
@Cacheable(cacheNames="todos",
    ↪ condition=#user.length < 10)
public List<Todo> retrieveTodos(String user) {
```

Annotazioni per il caching

- **@CachePut**: Aggiunge esplicitamente i dati alla cache
- **@CacheEvict**: Rimuove i dati dalla cache
- **@Caching**: Permette più annotazioni innestate sullo stesso metodo

2.5 Capitolo 7: Feature avanzate di Spring Boot

2.5.1 Configurazioni esternalizzate

Spring boot offre un modo flessibile e standardizzato per gestire i file di configurazione esterni.

Personalizzare i framework attraverso application.properties

Logging Alcune cose che possono essere personalizzate:

- Il path del file di configurazione per il logging
- Il path dell'output
- Il livello di logging

```
# Posizione del file di logging
  logging.config=
# Nome del log.
```

```
  logging.file=
# Configurare il livello di logging.
# Esempio ‘logging.level.org.springframework=TRACE‘
  logging.level.*=
```

Configurazione del server embedded Il server embedded di Spring è una delle caratteristiche più importanti. Tra le cose che si possono configurare:

- Porte
- Supporto SSL e configurazione
- Configurazione dell'access log

```
# Path dell'errore per i controller.
  server.error.path=/error
# Server HTTP port.
  server.port=8080
# Abilitare il supporto SSL.
  server.ssl.enabled=
# Percorso del key store per l'SSL
  server.ssl.key-store=
# Key Store Password
  server.ssl.key-store-password=
# Key Store Provider
  server.ssl.key-store-provider=
# Key Store Type
  server.ssl.key-store-type=
# Abilitazione del log degli accessi a Tomcat?
  server.tomcat.accesslog.enabled=false
# Numero massimo di connessioni accettate
  server.tomcat.max-connections=
```

Spring MVC Spring MVC può essere configurato tramite application.properties. Ad esempio:

```
# Formato della data ‘dd/MM/yyyy‘.
  spring.mvc.date-format=
# Locale da usare.
  spring.mvc.locale=
# Definire come il locale dovrebbe essere impostato.
  spring.mvc.locale-resolver=accept-header
# Bisognerebbe lanciare una "NoHandlerFoundException"
# nel caso l’handler non venisse trovato?
  spring.mvc.throw-exception-if-no-handler-found=false
# Prefisso per Spring MVC. Usato dal view resolver.
  spring.mvc.view.prefix=
# Suffisso per Spring MVC. Usato dal view resolver.
  spring.mvc.view.suffix=
```

Spring Starter Security

```
# Impostare vero per la sicurezza base
  security.basic.enabled=true
# Lista di path da mettere in sicurezza divisi da
    ↪ virgole
  security.basic.path=/**
# Lista di path da NON mettere in sicurezza divisi
    ↪ da virgole
  security.ignored=
# Nome dello user di base configurato da SPring
    ↪ Security
  security.user.name=user
# Password dell’utente base definito da Spring
  security.user.password=
# Ruolo dello user
  security.user.role=USER
```

Basi di dati

```
# Nome completo del driver JDBC
  spring.datasource.driver-class-name=
# Popolare il database usando ’data.sql’.
  spring.datasource.initialize=true
# Posizione JNDI della fonte dati.
  spring.datasource.jndi-name=
# Nome della fonte dati.
```

```
  spring.datasource.name=testdb
# Password di login del database.
  spring.datasource.password=
# Puntatore allo script di creazione DB.
  spring.datasource.schema=
# Db User per eseguire gli script
  spring.datasource.schema-username=
# Db password per eseguire gli scripts
  spring.datasource.schema-password=
# Url JDBC del database.
  spring.datasource.url=
# JPA - Inizializza lo schema all'avvio.
  spring.jpa.generate-ddl=false
# Use Hibernate's newer IdentifierGenerator for
    ↪ AUTO, TABLE and SEQUENCE.
  spring.jpa.hibernate.use-new-id-generator-mappings=
# Enable logging of SQL statements.
  spring.jpa.show-sql=false
```

Altre configurazioni Ad esempio:

- Profili
- Convertitori di messaggi HTTP (Jackson/JSON)
- Gestione delle transazioni
- Internazionalizzazione

```
# Comma-separated list (or list if using YAML) of
    ↪ active profiles.spring.profiles.active=
# HTTP message conversion. jackson or gson
  spring.http.converters.preferred-json-mapper=jackson
# JACKSON Date format string. Example 'yyyy-MM-dd
    ↪ HH:mm:ss'.
  spring.jackson.date-format=
# Default transaction timeout in seconds.
  spring.transaction.default-timeout=
# Perform the rollback on commit failures.
  spring.transaction.rollback-on-commit-failure=
```

```
# Internationalisation : Comma-separated list of
    ↪ basenames
  spring.messages.basename=messages
# Cache expiration for resource bundles, in sec. -1
    ↪ will cache for ever
  spring.messages.cache-seconds=-1
```

Proprietà personalizzate in application.properties

Consideriamo un esempio: vogliamo avere la possibilità di interagire con un servizio esterno. Vogliamo quindi poter esternalizzare la configurazione dell'URL di questo servizio.

Vogliamo quindi configurarlo in application.properties così:

```
somedataservice.url=http://abc.service.com/something
```

Vogliamo usare il valore di somedataservice.url nel nostro service.

```
@Component
public class SomeDataService {

  @Value("${somedataservice.url}")
  private String url;

  public String retrieveSomeData() {
    // Logic using the url and getting the data

    return "data from service";
  }
}
```

Notiamo:

- **@Component public class SomeDataService**: Il bean è gestito da Spring
- **@Value("$somedataservice.url")**: il calore di somedataservice.url sarà autowired all'url.

Configuration manager type-safe Nel caso si volessero impostare più elementi tramite properties ci sono due possibilità:

- Apporre l'annotazione @Value su tutti i campi
- Usare l'annotazione @ConfigurazionProperties("nome file di proprietà")

In maniera molto semplice il secondo definisce tutte le associazioni e poi vengono collegate all'interno del file.

```
@Component
@ConfigurationProperties("application")
public class ApplicationConfiguration {

  private boolean enableSwitchForService1;
  private String service1Url;
  private int service1Timeout;

  public boolean isEnableSwitchForService1() {
    return enableSwitchForService1;
  }

  public void setEnableSwitchForService1
    (boolean enableSwitchForService1) {
    this.enableSwitchForService1 =
      enableSwitchForService1;
  }

  public String getService1Url() {
    return service1Url;
  }

  public void setService1Url(String service1Url) {
    this.service1Url = service1Url;
  }

  public int getService1Timeout() {
    return service1Timeout;
  }
```

```
  public void setService1Timeout(int
      ↪ service1Timeout) {
    this.service1Timeout = service1Timeout;
  }
}
```

Notiamo:

- **@ConfigurationProperties("application")**: Seleziona il file di proprietà da utilizzare.
- Così configuriamo più bean in una volta sola
- Getter e Setter sono richiesti

In maniera molto semplice i valori sono salvati in application.properties:

```
application.enableSwitchForService1=true
application.service1Url=http://abc-dev.service.com/
  somethingelse
application.service1Timeout=250
```

Profili

Per ora abbiamo visto come esternalizzare i file di configurazione, ma se volessimo differenti valori per differenti ambienti?

I profili sono ciò che arrivano in aiuto. In application.properties:

```
spring.profiles.active=dev
```

Ora che abbiamo selezionato il profilo possiamo usare dei file di configurazione specializzati per profilo usando la notazione **application-profile-name.properties**.

Configurazione dei bean basata sul profilo Tutte le classi che sono annotate con @Component (o le sue versioni più specifi-

che) o @Configuration possono essere annotate anche con **@Profile(profilo)**.

Ad esempio:

```
// Dev
@Profile("dev")
@Configuration
public class DevSpecificConfiguration {

  @Bean
  public String cache() {
    return "Dev Cache Configuration";
  }
}

// Production

@Profile("prod")
@Configuration
public class ProdSpecificConfiguration {

  @Bean
  public String cache() {
    return "Production Cache Configuration -
        ↪ Distributed Cache";
  }
}
```

Altre opzioni per i valori di configurazione

Spring offre svariate possibilità per configurare le proprietà delle applicazioni. Ad esempio:

- Argomenti via linea di comando
- Creare delle proprietà di sistema col nome SPRING_APPLICATION_JSON e includere una configurazione JSON
- ServetConfig init

- ServletContext init
- Proprietà di Java (System.getProperties())
- Variabili di sistema
- Proprietà specifiche per il profilo
- Proprietà fuori dal .jar
- Proprietà all'interno del .jar

Configurazione YAML

YAML sta per "YAML Ain't Markup Language". È un formato leggibile ed è usato spesso per i file di configurazione. Per fare un esempio:

```
spring:
  profiles:
    active: prod
security:
  basic:
    enabled: false
  user:
    name=user-name
    password=user-password
  oauth2:
    client:
      clientId: clientId
      clientSecret: clientSecret
      authorized-grant-types:
          ↪ authorization_code,refresh_token,password
      scope: openid

application:
  enableSwitchForService1: true
  service1Url:
      ↪ http://abc-dev.service.com/somethingelse
  service1Timeout: 250
```

YAML permette anche la definizione di vari profili, sempre in maniera molto semplice:

```
application:
  service1Url: http://service.default.com
---
spring:
  profiles: dev
  application:
    service1Url: http://service.dev.com
---
spring:
  profiles: prod
  application:
    service1Url: http://service.prod.com
```

2.6 Capitolo 8: Spring Data

2.7 Capitolo 9: Spring Cloud

2.8 Capitolo 10: Spring Cloud Data Flow

2.9 Capitolo 11: Programmazione Reactive

2.10 Capitolo 12: Spring Best Practices

Capitolo 3

Introducing Spring Boot - Udemy

Spring boot - getting started.

3.1 Spring initializr

Spring Initializr, no, non è un typo.

Questo sito permette di creare velocemente un template che seleziona direttamente tutte le dipendenze.

In project Metadata abbiamo group, ovvero il nome del package di default, e artifact che sarà il nome del progetto, il tipo di wrapper (jar/war) e la versione del compilatore.

Nelle dipendenze possiamo dire quali dipendenze vogliamo. Ad esempio con questa configurazione otteniamo i file all'interno della cartella SpringInitializr.

Anche Spring CLI ci permette di eseguire la stessa funzione, per avere una lista delle possibilità scrivere "spring init -list" in una console. Per fare un esempio scriviamo "spring init -d=web my-app", dove -d sta per le dipendenze e my-app sarà il nome

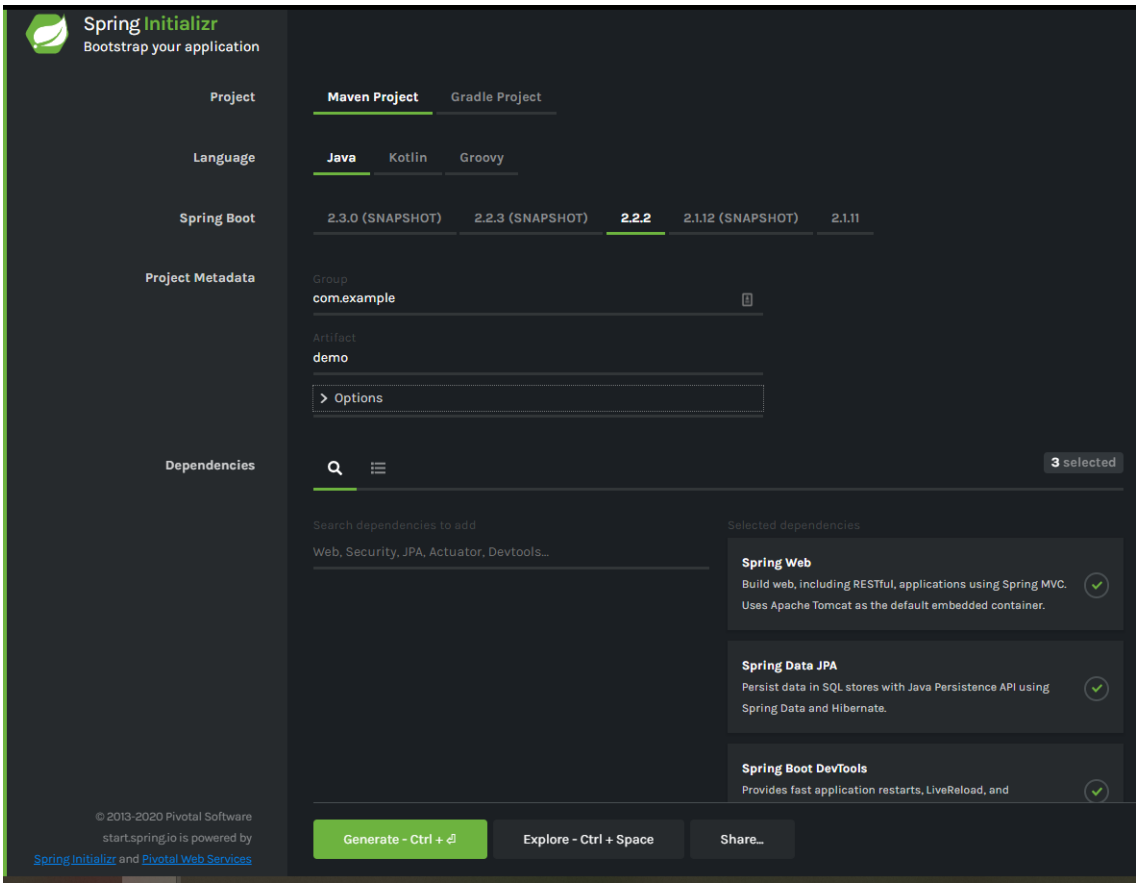

Figura 3.1: Spring Initializr

dell'applicazione.

Spring Initializr è integrato anche in molte IDE.

3.2 Java Build Tools

Un build tool è un sistema di scripting per automatizzare alcuni compiti. Tre tra i più importanti sono:

- Ant, ormai poco usato

- Maven. Offre la gestione delle dipendenze, è basato sui plugin e crea un file di configurazione XML
- Gradle. È basato su groovy.

Non c'è molta differenza tra Maven e Gradle.

3.2.1 Maven build

Aprendo il file pom.xml possiamo vedere la dichiarazione della versione di spring boot usata.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.2.2.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository
    ↪ -->
</parent>
```

Nella prima parte troviamo anche i nostri metadati:

```
<groupId>com.example</groupId>
<artifactId>demo</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo</name>
<description>Demo project for Spring
  ↪ Boot</description>
```

Vedremo anche la sezione delle dipendenze. Ad esempio:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa
      </artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
```

```
  </dependency>
</dependencies>
```

Notare che non abbiamo dovuto segnare alcuna versione per le dipendenze.

La versione di Java:

```
<properties>
  <java.version>1.8</java.version>
</properties>
```

Successivamente c'è la sezione per la build, in modo da definire il tipo di file wrapper.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot
        </groupId>
      <artifactId>spring-boot-maven-plugin
        </artifactId>
    </plugin>
  </plugins>
</build>
```

Infine, per eseguire il progetto Spring boot possiamo scrivere nella console "mvn spring-boot:run"

Starter POM

È molto facile crearne uno, qua la documentazione e qua una spiegazione più discorsiva.

Ad esempio possiamo vedere tra le dipendenze "spring-boot-starter-web", per importare un ambiente fullstack completo di tomcat e spring web MVC.

3.2.2 Gradle build

Il file di build è build.gradle.

Differentemente da Maven, non abbiamo un superparent da cui importare le impostazioni di default.

All'inizio viene fissata la versione si SpringBoot e dei vari plugin:

```
plugins {
  id 'org.springframework.boot' version
     ↪ '2.2.2.RELEASE'
  id 'io.spring.dependency-management' version
     ↪ '1.0.8.RELEASE'
  id 'java'
}
```

Le dipendenze possono essere importate velocemente:

```
dependencies {
  implementation 'org.springframework.boot:
     ↪ spring-boot-starter-data-jpa'
  implementation 'org.springframework.boot:
     ↪ spring-boot-starter-web'
  developmentOnly 'org.springframework.boot:
     ↪ spring-boot-devtools'
  testImplementation('org.springframework.boot:
     ↪ spring-boot-starter-test') {
    exclude group: 'org.junit.vintage',
    module: 'junit-vintage-engine'
  }
}
```

3.3 DevTools e live reload

DevTools non viene incluso all'interno del jar finale. Dev tools offre alcuni sturmenti utili durante la fase dello sviluppo, ad esempio il mantenimento della cache (spring.thymeleaf.cache)

per velocizzare l'app, ha sistemi di restart automatico ogni volta che viene compilata una classe (spring loaded e JRebel). C'è anche un live reload, ovvero un estensione browser che aggiorna la pagina ad ogni cambiamento lato server.

3.4 Spring boot

Iniziamo scrivendo un controller in un file groovy. Il file HelloWorld.groovy si occuperà semplicemente di mostrare la stringa "Hello, World!" sul browser. Qua il codice.

3.5 Executable JAR

I JAR sono archivi che contengono le classi compilate con tutte le loro dipendenze. Sono cloud friendly, nel senso che possono essere direttamente caricate online sul server.

Java non offre nessun metodo diretto per caricare file jar innestati. Spring boot offre anche questa funzione.

Maven: "mvn package"
Gradle: "gradle boot"

Per creare il jar scrivere "spring jar [nome jar] [file da includere]"

Capitolo 4

Spring Core

Learn Spring Framework 4 and Spring Boot

Spring Core - Learn Spring Framework 4 and Spring Boot Sito dell'istruttore

Le caratteristiche principali di spring sono:

- Beans: Spring usa POJO (plain Old Java Objects) o "Spring beans"
- Dependecy injection (DI):
- Inversion of Control (IoC)

4.1 Hello World

Codice

```
@Component
public class HelloWorld {
    public void sayHello(){
        System.out.println("Hello world!");
    }
}
```

Possiamo notare che la nostra classe HelloWorld è registrata come component. Questo vuol dire che Spring interpreterà la classe come un bean. La cosa risulta utile all'interno dell'applicazione:

```
@SpringBootApplication
public class HelloworldApplication {
    public static void main(String[] args) {
        ApplicationContext ctx = SpringApplication
          .run(HelloworldApplication.class, args);

        HelloWorld helloWorld = (HelloWorld)
            ↪ ctx.getBean("helloWorld");
        helloWorld.sayHello();
    }
}
```

Analizziamo un attimo questo codice:

- @SpringBootApplication: Permette all'applicazione di attivare tre funzioni automaticamente:
  - @EnableAutoConfiguration: Cerca di configurare automaticamente Spring in base alle dipendenze inserite
  - @ComponentScan: cerca automaticamente i @Component nel package dell'applicazione
  - @Configuration: Permette di registrare beans extra nel contesto o importare classi di configurazione addizionali
- ApplicationContext: serve per ricevere le configurazioni dell'applicazione
- ctx.getBean: Spring cerca tra i vari bean (component) uno che abbia quel nome e lo passa come oggetto
- (HelloWorld): il casting serve perchè altrimenti l'application context passerebbe un oggetto

4.2 Dependecy Injection

È una della componenti principali di Spring. L'applicazione si occupa di collegare automaticamente gli oggetti ai loro beans. Questo approccio, ovvero partire dall'oggetto dipendente per risalire, è il concetto dell'IoC. Il lato positivo dell'IoC è il poter usare diversi bean a runtime e riduce la coesione del codice, in modo da poter rendere più indipendenti le classi.

Ci sono più tipi di dependecy injection:

- **Basata sui costruttori**. Preferito per le classi che non possono essere istanziate senza le loro dipendenze
- **Basata sui setter**. Preferito nelle applicazioni di Spring, più flessibile dei costruttori.
- **Basata sulle interfacce**. È considerata la best practice. È allineata con i principi della programmazione orientata agli oggetti. Permette flessibilità nella composizione delle classi.

Per capire meglio come usare autowired andare a pagina 25.

4.2.1 Profili

Codice.

Abbiamo creato un controller che si occupa di generare un'istanza di HelloWorldService e restituire il suo saluto.

```
@Controller
public class GreetingController {

    private HelloWorldService helloWorldService;

    @Autowired
    public void setHelloWorldService
        ↪ (HelloWorldService helloWorldService) {
        this.helloWorldService = helloWorldService;
```

```
    }

    public String sayHello(){
        String greeting =
            ↪ helloWorldService.getGreeting();

        System.out.println(greeting);

        return greeting;
    }
}
```

Possiamo riconoscere la sua natura dall'annotazione Controller. QUando viene creato un GreetingController viene attivato Autowired su setHelloWorldService e verrà creato uno dei componenti disponibili.

```
// Interfaccia
public interface HelloWorldService {
    public String getGreeting();
}

//_______________________________
//HelloWorldServiceEnglishImpl

@Component
@Profile("english")
public class HelloWorldServiceEnglishImpl implements
    ↪ HelloWorldService {

    @Override
    public String getGreeting() {
        return "Hello world";
    }
}

//_______________________________
//HelloWorldServiceSpanishImpl

@Component
@Profile("spanish")
```

```
public class HelloWorldServiceSpanishImpl implements
    ↪ HelloWorldService{

    @Override
    public String getGreeting() {
        return "Hola Mundo";
    }
}
```

Come fa il controller a distinguere tra le varie versioni e sapere quale implementare? Notare che nelle due implementazioni abbiamo l'annotazione Profile con un tag. Nel file .properties abbiamo creato appositamente una stringa "spring.profiles.active=english" per segnalare al controller che il profilo che vogliamo in questo momento è quello inglese.

4.2.2 Default profiles

Annotare un bean con "@Profile("default")" viene automaticamente riconosciuto da Spring come un profilo speciale, ovvero quello di base se non è presente alcuna annotazione nel file di proprietà.

Per poter usare una classe sia come profilo di defult che come classe normale si possono usare più profili, basta scrivere all'interno delle annotazioni tutti i nomi separati da virgole e racchiunderli tra graffe.
E.g. @Profile("it", "default")

4.3 Spring Java Configuration

4.3.1 Component scan

Come detto precedentemente, Component scan funziona cercando i vari bean all'interno del suo package. Se volessimo dire di cercare anche altri bean all'interno di altri package do-

vremmo scrivere tra le annotazioni sopra la classe "@ComponentScan("[nome package]")".

Attenzione!: quando si chiede un bean al contesto il nome deve iniziare con la minuscola, anche se la classe ha un nome che inizia con una maiuscola.

4.3.2 Spring Java Configuration classes

```
@Configuration
public class HelloConfig {

    @Bean
    @Profile("default, english")
    public HelloWorldService
        ↪ helloWorldServiceEnglish(){
        return new HelloWorldServiceEnglishImpl();
    }

    @Bean
    @Profile("spanish")
    public HelloWorldService
        ↪ helloWorldServiceSpanish(){
        return new HelloWorldServiceSpanishImpl();
    }
}
```

Possiamo svolgere la stessa funzione creando un file di configurazione per istanziare tutti i bean. Le annotazioni da usare sono Configuration, Bean e Profile come visto precedentemente.

Perchè usare un file di configurazione? È usato di solito con librerie di terze parti perchè non avendo accesso al codice sorgente ci si deve adattare.

4.3.3 Factory beans

Codice.

Si usa solitamente quando ci si deve collegare ad un database.

```
public class HelloWorldFactory {

   public HelloWorldService
     createHelloWorldService(String language){
       HelloWorldService service = null;

       switch (language){
           case "en":
               service = new
                   ↪ HelloWorldServiceEnglishImpl();
               break;
           case "es":
               service = new
                   ↪ HelloWorldServiceSpanishImpl();
               break;
           case "fr":
               service = new
                   ↪ HelloWorldServiceFrenchImpl();
               break;
           case "de":
               service = new
                   ↪ HelloWorldServiceGermanImpl();
               break;
           case "pl":
               service = new
                   ↪ HelloWorldServicePolish();
               break;
           case "ru":
               service = new
                   ↪ HelloWorldServiceRussianImpl();
               break;
           default:
             new HelloWorldServiceEnglishImpl();
       }

       return service;
   }
}
```

Abbiamo creato una factory che in base alla configurazione scelta inizializza un bean differente. Questo viene effettuato passando attraverso il file di configurazione:

```
@Configuration
public class HelloConfig {

    @Bean
    public HelloWorldFactory helloWorldFactory(){
        return new HelloWorldFactory();
    }

    @Bean
    @Profile("english")
    public HelloWorldService
        ↪ HelloWorldServiceEnglish(HelloWorldFactory
        ↪ factory){
        return factory.createHelloWorldService("en");
    }
}
```

Come possiamo vedere ora Il file di configurazione crea una Factory che viene passata all'istanziazione dell'HelloWorldService.

4.3.4 Opzioni avanzate per Autowire

```
@Configuration
public class HelloConfig {

    @Bean
    public HelloWorldFactory helloWorldFactory(){
        return new HelloWorldFactory();
    }

    @Bean
    @Profile("english")
    @Primary
    public HelloWorldService
        ↪ helloWorldServiceEnglish(HelloWorldFactory
        ↪ factory){
```

```
        return factory.createHelloWorldService("en");
    }

    @Bean
    @Profile("spanish")
    public HelloWorldService
        ↪ helloWorldServiceSpanish(HelloWorldFactory
        ↪ factory){
        return factory.createHelloWorldService("es");
    }
}
```

L'annotazione Primary segnala a Spring che se non viene chiesto niente allora il bean da preferire è quello.

Si può anche usare l'injection in maniere più elaborate, ad esempio:

```
@Configuration
public class HelloConfig {

    @Bean
    public HelloWorldFactory helloWorldFactory(){
        return new HelloWorldFactory();
    }

    @Bean
    @Profile("english")
    @Primary
    public HelloWorldService
        ↪ helloWorldServiceEnglish(HelloWorldFactory
        ↪ factory){
        return factory.createHelloWorldService("en");
    }

    @Bean
    @Profile("spanish")
    public HelloWorldService
        ↪ helloWorldServiceSpanish(HelloWorldFactory
        ↪ factory){
        return factory.createHelloWorldService("es");
```

```
    }


    @Bean("french")
    public HelloWorldService
        ↪ helloWorldServiceFrench(HelloWorldFactory
        ↪ factory){
        return factory.createHelloWorldService("fr");
    }

    @Bean
    public HelloWorldService
        ↪ helloWorldServiceDeutsche(HelloWorldFactory
        ↪ factory){
        return factory.createHelloWorldService("de");
    }
}

// ____________________________________________

//____________________________________________

private HelloWorldService helloWorldService;

    private HelloWorldService helloWorldServiceGerman;

    private HelloWorldService helloWorldServiceFrench;


  // Wired tramite Primary
    @Autowired
    public void setHelloWorldService
        ↪ (HelloWorldService helloWorldService) {

        this.helloWorldService = helloWorldService;
    }

  // Wired tramite identifier (nome del bean)
    @Autowired
    @Qualifier("helloWorldServiceGerman")
    public void setHelloWorldServiceGerman
```

```
        ↪ (HelloWorldService
        ↪ helloWorldServiceGerman) {
        this.helloWorldServiceGerman =
          helloWorldServiceGerman;
    }

  // Wired tramite nome del bean
  // (equivalente a Qualifier)
    @Autowired
    @Qualifier("french")
    public void setHelloWorldServiceFrench
        ↪ (HelloWorldService
        ↪ helloWorldServiceFrench) {

        this.helloWorldServiceFrench =
            ↪ helloWorldServiceFrench;
    }
```

4.4 Configurazione tramite XML

IntelliJ aiuta già dando tra le opzioni anche la creazione di un file apposito per spring. Qua inseriremo prima di tutto la tag per la component scan all'interno della quale indicheremo il/i nostro/i package per la ricerca dei bean.

All'interno del main inseriremo il tag @ImportResource("classpath:[locazione sotto la cartella resources per il file di configurazione]")

Cose fighe: se abbiamo una factory per i bean non dichiariamo direttamente il bean con la sua classe ma in questo modo:

```
<bean id="helloWorldServiceFactory"
  class="com.example.springtest.demo
  .service.HelloWorldServiceFactory" />

<bean id="italian"
    ↪ factory-bean="helloWorldServiceFactory"
    ↪ factory-method="helloWorldServiceCreate">
  <constructor-arg value="it" />
```

```
</bean>
```

Un vantaggio dei file di configurazione è che se ne possono creare di più da integrare in parti separate del progetto tramite @ImportResource. In questo modo si possono personalizzare i valori di default e le configurazioni in base alla sezione.

Siti utili

Documentazione Spring boot
Documentazione Spring API
Piattafroma Spring IO
Guide
Spring CLI
SpringInitializr
Per portare velocemente risorse nel progetto. Quando si aggiunge una dipendenza bisogna inserirla in pom.xml e poi fare mvn install.
Lombok.
Per gestire versioni parallele di più ambienti di sviluppo
Postman.
Swagger UI