SUPSI

Spring MVC e pagine dinamiche con Thymeleaf

Spring Boot static content

- Per default Spring Boot serve il contenuto statico (pagine HTML, file CSS, javascript, immagini, etc.) dalla cartella ***static***.

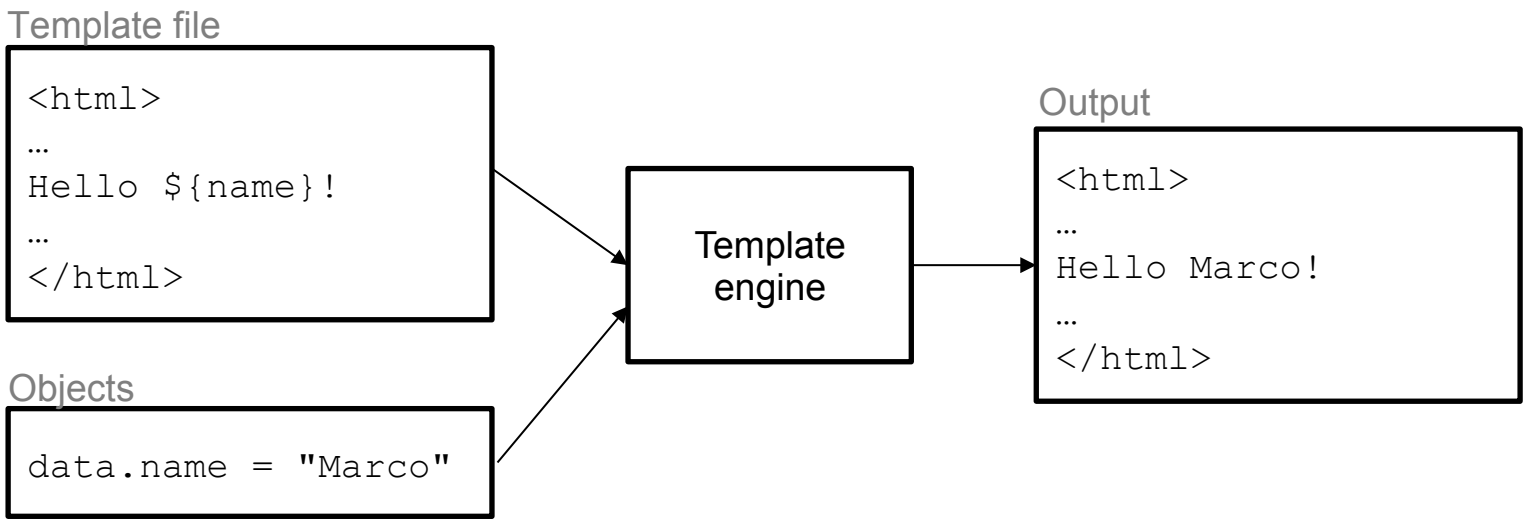
- Utilizza la `ResourceHttpRequestHandler` di Spring MVC e quindi se ne può modificare il comportamento.

- Con la proprietà `spring.resources.static-locations` è possibile personalizzare dove le risorse statiche vengono cercate.

- Questa è la configurazione di default:
  - **`spring.resources.static-locations`**`=classpath:/META-INF/resources/,classpath:/resources/,classpath:/static/,classpath:/public/`

Template engine (contenuto dinamico)

- Un template engine è uno strumento per **generare file basati su modelli**.
- I **template engine** nascono per favorire una maggiore separazione tra la parte di funzionalità di un'applicazione web e la parte di presentazione. Due ambiti gestiti spesso da soggetti diversi.



Template engines in Spring Boot

- Spring Boot include auto-configurazioni per:
  - FreeMarker
  - Groovy
  - Mustache
  - Thymeleaf

- JSP non sono supportate di default in Spring Boot (ci sono alcune limitazioni, soprattutto quando si usa Tomcat embedded)

- I templates sono cercati nella cartella **src/main/resources/templates**

Properietà template engine

- È possibile configurare ogni template engine supportato da Spring Boot
- Per esempio, le configurazioni di Thymeleaf:

```
spring.thymeleaf.cache=true # Enable template caching.

spring.thymeleaf.enabled=true # Enable MVC Thymeleaf view
resolution.

spring.thymeleaf.encoding=UTF-8 # Template encoding.

spring.thymeleaf.mode=HTML # Template mode to be applied to
templates.

spring.thymeleaf.suffix=.html # Suffix that gets appended to view
names when building a URL.

spring.thymeleaf.prefix=classpath:/templates/ # Prefix that gets
prepended to view names when building a URL.
```

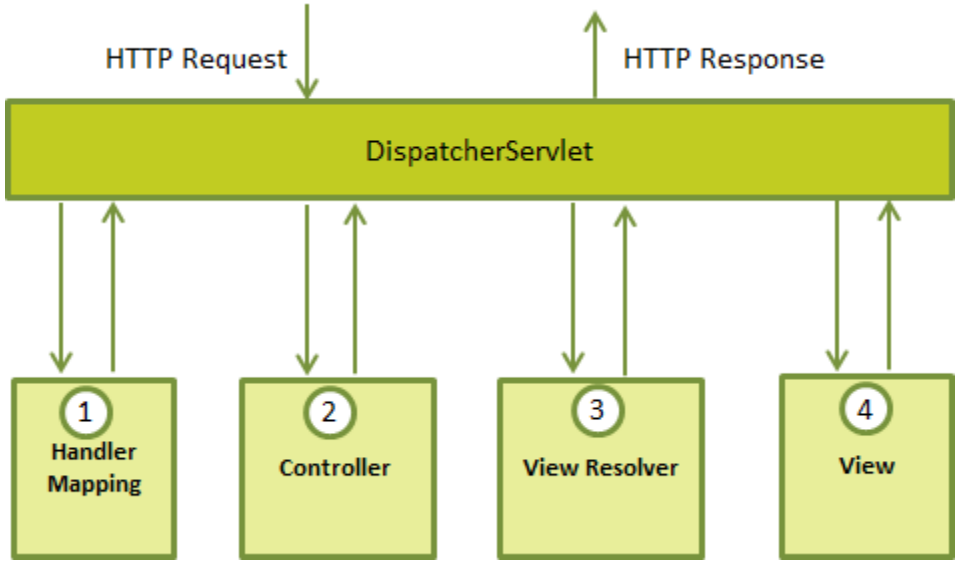
Spring MVC

- Implementa perfettamente il pattern MVC:
  - i **Model** sono rappresentati dalle classi che a loro volta rappresentano gli oggetti gestiti e le classi di accesso al database
  - le **View** sono rappresentate dai vari file che vengono compilati in HTML e da eventuali classi per l'esportazione in formati diversi da HTML (PDF, XLS, CSV…)
  - i **Controller** sono rappresentati da classi (chiamate appositamante Controller) che rimangono "in ascolto" su un determinato URL e, grazie ai Model e alle View, si occupano di gestire la richiesta dell'utente.

Spring MVC DispatcherServlet

- È disegnato attorno ad una *DispatcherServlet* che gestisce tutte le richieste e risposte HTTP.

- Il flusso di elaborazione di una richiesta nel *DispatcherServlet* è illustrato nello schema seguente:



Spring MVC DispatcherServlet

- Cosa succede quando una richiesta arriva al *DispatcherServlet?*
  - Il *DispatcherServlet* consulta l'oggeto *HandlerMapping* per chiamare il **controller** appropriato (mappatura URL → metodo di una classe)
  - Il controller prende la richiesta e chiama i metodi appropriati a seconda che sia una richiesta di tipo GET o POST. Il metodo crea un **modello** basato sulla logica di business (crea degli oggetti) e ritorna il nome della **view** da chiamare al DispatcherServlet
  - Il *DispatcherServlet*, tramite l'oggetto *ViewResolver*, sceglie la view definite per la richiesta
  - Appena la view è finalizzata, il *DispatcherServlet* passa i dati del modello alla view, che è finalmente "renderizzata" sul browser

@Controller

- L'annotazione `@Controller` indica che una classe particolare serve al ruolo di controller.

```
@Controller
public class PersonController {

   @GetMapping("/")
   public String index(Model model){
      model.addAttribute("persons", persons);
      return "index";
   }
```

- `@GetMapping` è una shortcut per `@RequestMapping(method = RequestMethod.GET)`

Return types supportati

- **`String`** il valore è interpretato come il nome della view da caricare. Il modello può essere arricchito dichiarando un `Model` come argomento del metodo

- Se il metodo è annotato con **`@ResponseBody`** allora l'oggeto ritornato dal metodo è scritto nel body della risposta, utilizzando un serializzatore.

- **`ModelAndView`** la view e il modello sono implicitamente passati nel costruttore

- Le possibilità sono molte:

  – https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-ann-return-types

@RequestParam

- L'annotazione `@RequestParam` indica che un parametro di un metodo dev'essere associato a un parametro della richiesta.

- È associato sia ai parametri della query string, che ai parametri passati nel corpo della richiesta (ma solo quando `Content-Type = application/x-www-form-urlencoded`).

```
@PostMapping("/person/new")
public String submit(@RequestParam String name, Model model) {
    model.addAttribute("person", new Person(name));
    return "personView";
}
```

Binding parametri-oggetto

- Quando si hanno tanti parametri nella richiesta HTTP, piuttosto che aggiungere tanti `@RequestParam` è più comodo utilizzare un oggetto apposito (in questo esempio di tipo `Person`)

```
@PostMapping("/person/new")
public String submit(Person person, Model model) {
    model.addAttribute("person", person);
    return "personView";
}
```

- In questo caso, i valori dei campi nell'oggetto `Person` saranno popolati leggendo i parametri della richiesta. Se la richiesta ha, per esempio, il parametro `name=Marco`, il campo `name` dell'oggetto avrà il valore Marco. Questa operazione è chiamata **binding**.

@ModelAttribute

- L'annotazione `@ModelAttribute`, quando utilizzato come argomento di metodo, indica che il valore dell'argomento deve essere associato ad un attributo del modello, e sarà quindi possibile usarlo nella visualizzazione web.

```
@PostMapping("/person/new")
public String submit(@ModelAttribute("person") Person person) {
    // codice che usa in qualche modo l'oggetto person
    return "personView";
}
```

- Senza esplicitamente aggiungere l'oggetto al modello (usando `model.addAttriubte`), questo sarà aggiunto come attributo al modello.

Gestire i redirect

- Se il nome di una view inizia con il prefisso `redirect:` il View Resolver lo riconoscerà come indicazione che un redirect è necessario.

- Il resto del nome della view sarà riconosciuto come l'URL di redirezione.

```
@PostMapping("/person/new")
public String post(@ModelAttribute("person") Person person){
  //
  return "redirect:/";
}


@PostMapping("/person/{id}/edit")
public String put(@PathVariable int id,  @ModelAttribute("person") Person person){
//
return "redirect:/person/{id}";
}
```

Views

- In Spring i ViewResolver consentono di rappresentare i modelli in un browser (in HTML) senza legarsi a una specifica tecnologia di visualizzazione.

- In Spring Boot decidere se usare Freemarker piuttosto che Thymeleaf è solo una questione di configurazione.

- Per esempio, per usare Thymeleaf basta aggiungere la dipendenza nel pom.xml:

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

Dipendenze template engines

- Freemarker

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

- Groovy

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

Thymeleaf

- Thymeleaf è un moderno server-side template engine scritto in Java
- Il suo principale obiettivo è di creare **“elegant natural templates”** − ossia: l’HTML può essere visualizzato correttamente nei browser anche come prototipi statici, consentendo una maggiore collaborazione nel team di sviluppo.
- Freemarker

```
<html>
…
Hello ${name}!
…
</html>
```

Thymeleaf

```
<html>
…
Hello <span th:text="${name}">Marco</span>!
…
</html>
```

- Visualizzato direttamente in un browser Thymeleaf non espone mai il suo codice

```
Hello ${name}!
```

```
Hello Marco!
```

Thymeleaf: esempio

```
<h1 th:text="${product.name}">Titolo</th>

<table>
  <tbody>
    <tr th:each="prod: ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="${#numbers.formatDecimal(prod.price, 1, 2)}">0.99</td>
    </tr>
  </tbody>
</table>
```

th:text

- L'attributo **th:text** rimpiazza il contenuto di un tag.

```
<p th:text="${msg.welcome}">Welcome everyone!</p>
```

- Se `msg.getWelcome()` ritorna la stringa `"Ciao"` allora il risultato sarà:

```
<p>Ciao</p>
```

Attributi di un tag

- In generale se si vuole rendere dinamico un attributo di un elemento HTML è sufficiente aggiungere **th:** all'attributo.
- Per esempio:

```
<img th:alt="${info}">

<p th:title="${product.title}" />
```

th:each

- **th:each** ripete il tag tante volte quanti sono gli elementi nell'array o la lista ritornata dall'espressione

```
<tr th:each="book : ${books}">
    <td th:text="${book.id}">ID</td>
    <td th:text="${book.title}">Title</td>
</tr>
```

- Se books fosse una lista di due oggetti Book:

```
<tr>
    <td>1</td>
    <td>Titolo 1</td>
</tr>
<tr>
    <td>2</td>
    <td>Titolo 2</td>
</tr>
```

th:href, th:src, th:action

- Tutte le espressioni che iniziano con @{} son dette **link expressions** e servono per creare degli URL e sono molto utili perché possono aggiungere il contesto nel quale è l'applicazione

- Se la mia applicazione è pubblicata in tomcat potrebbe avere un URL http://host:8080/miapplicazione/ dove *miapplicazione* è il nome del contesto

- Un espressione così:

```
<a th:href="@{/order/list}">...</a>
```

- Sarebbe convertita in

```
<a href="/miapplicazione/order/list">...</a>
```

th:href, th:src, th:action

- Link expression funzionano perfettamente anche per gli attributi src (nel tag img per esempio) o action (nelle form).

- È possibile creare il link dinamicamente, per esempio:

```
<a th:href="@{'/person/'+${person.id}}">dettaglio</a>
```

th:if

- Utilizzando th:if è possibile porre una condizione per la visualizzazione di un tag

```
<div th:if="${user.isAdmin()}"> ...

<div th:if="${variable.something} == null"> ...
```

- In questo caso solo se l'utente è amministratore il primo div sarà visualizzato
- E solo se varible.something sarà uguale a null il secondo div sarà visualizzato

th:object

- L'attributo **th:object** permette di semplificare l'accesso ai campi di un oggetto in un determinato contesto
- Per esempio:

```
<form>
        <input th:value="${person.address.line1}">
        <input th:value="${person.address.country}”>
</form>
```

- Può essere riscritto

```
<form th:object="${person.address}">
        <input name="line1" th:value="*{line1} ">
        <input name="country" th:value="*{country}">
</form>
```

th:field

- Per aggiungere un input in un form:

```
<input type="text" th:field="*{firstname}" />
```

- L'attributo **th:field** si comporta diversamente a dipendenza se è incluso in un tag `input`, `select` o `textarea`.

- Per esempio la linea sopra è simile a:

```
<input type="text" id="firstname" name="firstname" th:value="*{firstname}" />
```

Expression Utility Objects

- Thymeleaf offre un insieme di oggetti utility che ci possono aiutare in compiti comuni e ricorrenti:
  - **#dates**: utility methods for **java.util.Date** objects: formatting, component extraction, etc.
  - **#calendars**: analogous to **#dates**, but for **java.util.Calendar** objects.
  - **#numbers**: utility methods for formatting numeric objects.
  - **#strings**: utility methods for **String** objects: contains, startsWith, prepending/appending, etc.
  - **#objects**: utility methods for objects in general.
  - **#bools**: utility methods for boolean evaluation.
  - **#arrays**: utility methods for arrays.
  - **#lists**: utility methods for lists.
  - **#sets**: utility methods for sets.
  - **#maps**: utility methods for maps.

Expression Utility Objects

```
<p th:text="${#dates.format(date, 'dd-MM-yyyy HH:mm')}"></p>

<p th:text="${#numbers.formatDecimal(num,2,3)}"></p>


<p th:text="${#strings.isEmpty(string)}"></p>

<p th:text="${#strings.substring(name,3,5)}"></p>

<p th:text="${#strings.replace(name,'las','ler')}"></p>


<span th:text="${#lists.size(prod.comments)}">2</span>

<p th:if="${not #lists.isEmpty(prod.comments)}">commenti</p>
```

Acceso a metodi statici

- Per accedere a metodi statici si può usare **T()**, per esempio per accedere alla constante `MAX_TESTCASE_VALUE_LENGTH` della classe `ch.supsi.utils.Constants`

```
<td>
 <input class="form-control" type="text" name="inputValue"
        th:maxlength="${T(ch.supsi.utils.Constants).MAX_TESTCASE_VALUE_LENGTH}" />
</td>
```

Esempio: model

```
public class Greeting {

    private long id;
    private String content;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }
}
```

In questo esempio il modello non è altro che una istanza della classe *Greeting*.

Contiene due campi: id e content.

Esempio: controller

```
@Controller
public class GreetingController {

    @RequestMapping(value="/greeting", method=RequestMethod.GET)
    public String greetingForm(Model model) {
        model.addAttribute("greeting", new Greeting());
        return "greeting";
    }

    @RequestMapping(value="/greeting", method=RequestMethod.POST)
    public String greetingSubmit(@ModelAttribute Greeting greeting, Model model) {
        model.addAttribute("greeting", greeting);
        return "redirect:/greeting";
    }

}
```

Nel metodo che gestisce la richiesta in POST, Spring MVC cerca di mappare i parametri ricevuti dalla richiesta in campi dell'oggetto greeting.

Spring MVC: view

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Getting Started: Handing Form Submission</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    <h1>Form</h1>
    <form action="#" th:action="@{/greeting}" th:object="${greeting}" method="post">
          <p>Id: <input type="text" th:field="*{id}" /></p>
        <p>Message: <input type="text" th:field="*{content}" /></p>
        <p><input type="submit" value="Submit" /></p>
    </form>
</body>
</html>
```

Da notare la dichiarazione del namespace th con **xmlns:th=http://www.thymeleaf.org**

Link utili

- http://spring.io/guides/gs/serving-web-content/
- http://spring.io/guides/gs/handling-form-submission/
- https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html
- https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-developing-web- applications.html
- http://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html
- http://www.thymeleaf.org/doc/articles/standarddialect5minutes.html