

# Programmazione avanzata

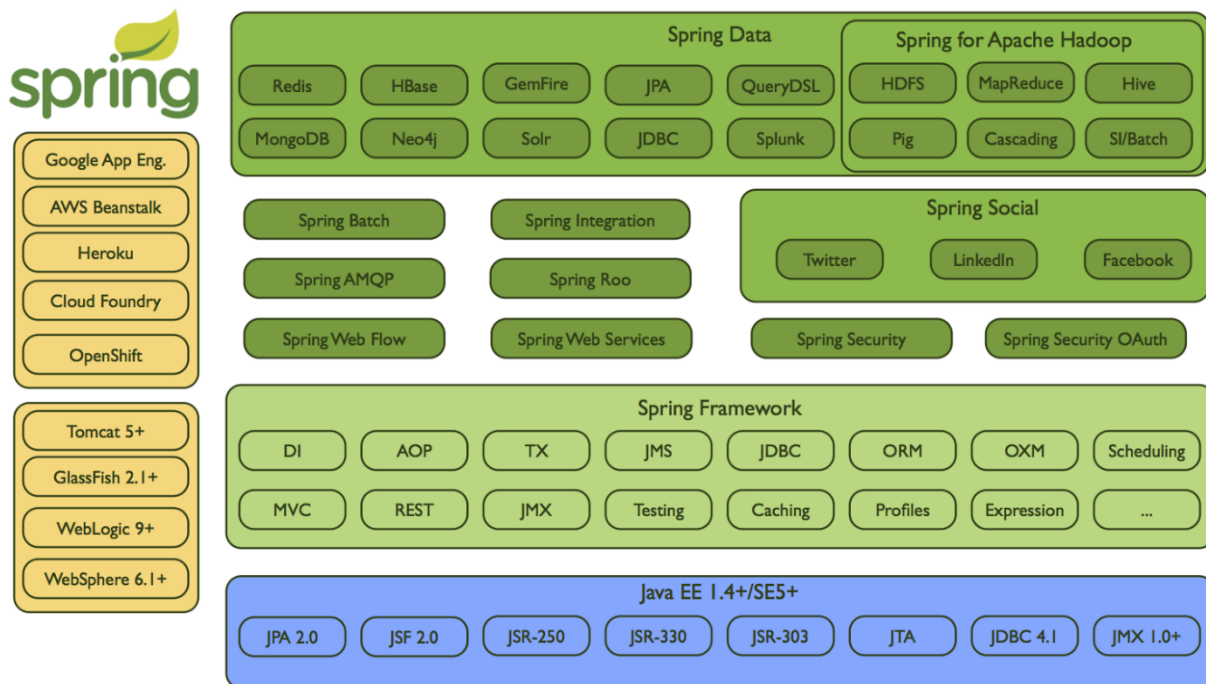
## Lezione 7

Creazione di un'applicazione Web utilizzando il framework Spring

### Il framework Spring

Spring è un framework opensource java per la realizzazione di applicazioni web basate sul modello Java EE (Enterprise Edition). Le applicazioni Java EE sono delle applicazioni che a differenza delle applicazioni Java SE (Standard Edition) mostrano un'interfaccia non grafica o testuale all'utente ma basata sul web.

Spring è una composizione di una serie di diversi progetti, ciascuno dei quali implementano una serie di funzioni. La flessibilità del sistema e la sua ricchezza fanno Spring un riferimento per la realizzazione di applicazioni e servizi web.

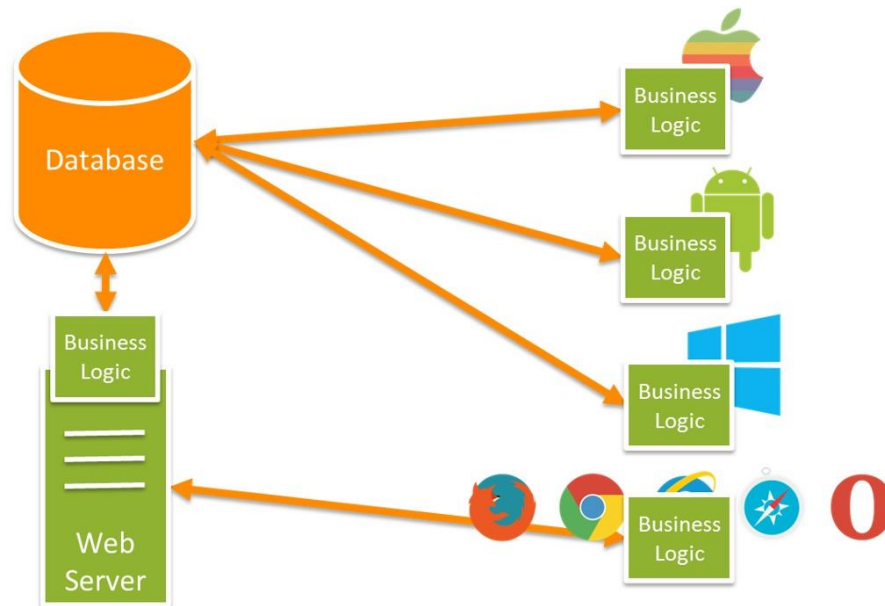


Tra le varie componenti del framework vale la pena di menzionare:

- Spring Boot, che serve per la creazione dello scheletro di un progetto
- Spring MVC, che serve per la realizzazione di applicazioni web
- Spring Data JPA, che serve per interagire in maniera semplificata con un database

## Creazione del progetto con Spring Boot e prima esecuzione

L'applicazione che vogliamo realizzare è un'applicazione che espone direttamente un'interfaccia web che può essere utilizzata dagli utenti attraverso un browser:



Lo scheletro di una prima applicazione web con tutte le dipendenze può essere creato utilizzando Spring Boot, che offre un'interfaccia web per la creazione dello scheletro di un'applicazione all'indirizzo <https://start.spring.io>

L'immagine mostra lo screenshot dell'interfaccia web di Spring Initializr (<https://start.spring.io>). L'interfaccia è divisa in tre sezioni principali: Project, Language e Dependencies.

**Project:** ☒ Maven Project, ☐ Gradle Project

**Language:** ☒ Java, ☐ Kotlin, ☐ Groovy

**Spring Boot:** ☐ 3.0.0 (SNAPSHOT), ☐ 3.0.0 (M4), ☐ 2.7.4 (SNAPSHOT), ☒ 2.7.3, ☐ 2.6.12 (SNAPSHOT), ☐ 2.6.11

**Project Metadata:**

Group	it.unipi
Artifact	PrimaAppWeb
Name	PrimaAppWeb
Description	Prima App Web con Spring Boot
Package name	it.unipi.PrimaAppWeb

**Dependencies:**

**Spring Web** **WEB**  
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

**Spring Boot DevTools** **DEVELOPER TOOLS**  
Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

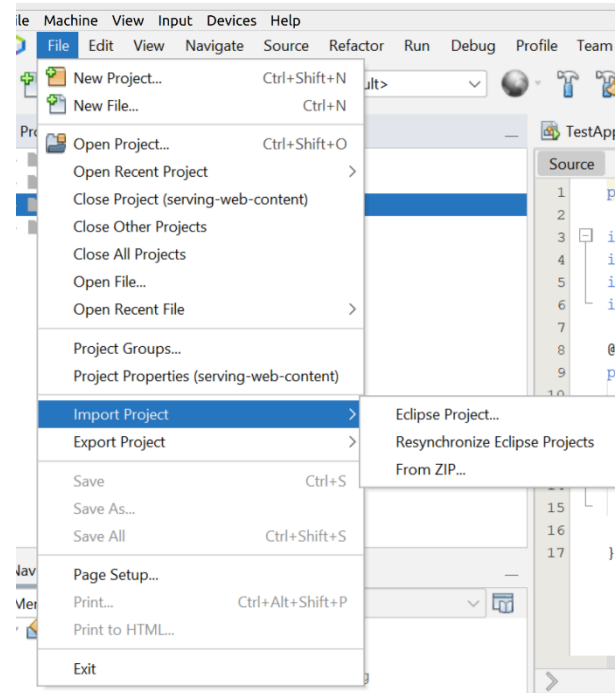
**Thymeleaf** **TEMPLATE ENGINES**  
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

Nel nostro caso creiamo un progetto di tipo **Maven, Java 11**, con le seguenti tre dipendenze:

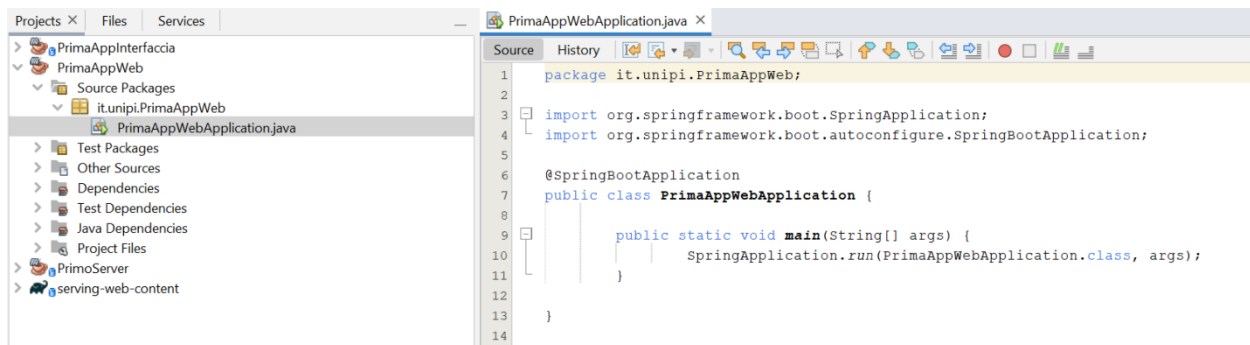
- Spring Web, utilizzato per creare applicazioni web
- Spring Boot DevTools, che fornisce strumenti utili alla realizzazione dell'applicazione
- Thymeleaf, che serve per realizzare in maniera semplice l'inclusione dei dati all'interno delle pagine web

Una volta specificate le dipendenze si può chiedere all'applicazione web di creare lo scheletro della nostra applicazione, la quale può essere scaricata sotto forma di archivio zip.

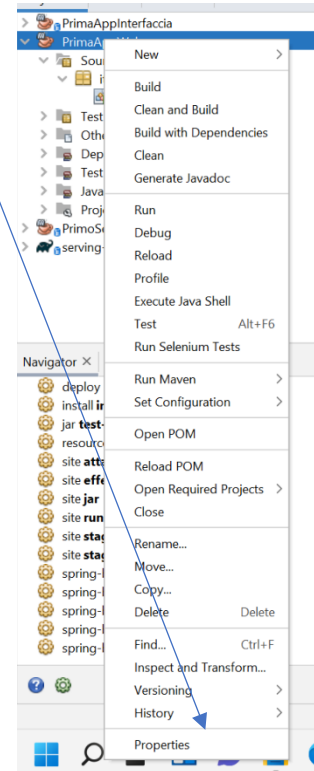
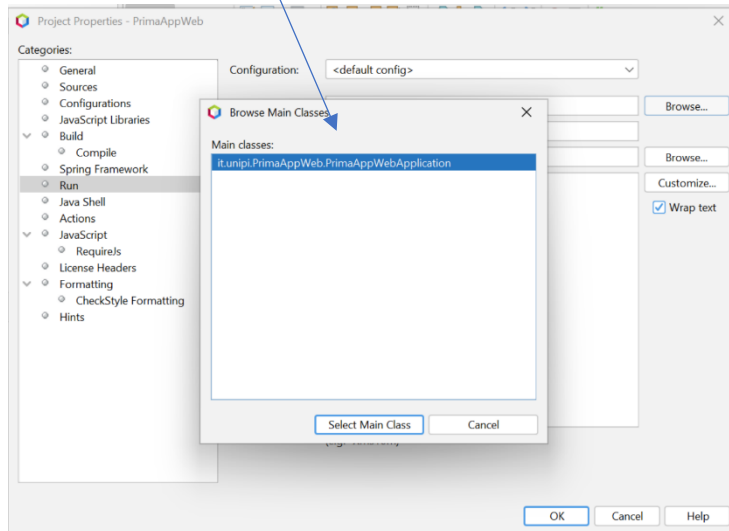
Una volta scaricato il pacchetto questo può essere importato in NetBeans attraverso la funzione File->Import Project-> From ZIP



Una volta importato il progetto bisogna effettuare l'operazione di 'Build' in maniera tale da permettere al sistema di scaricare le dipendenze. Una volta effettuata la prima "Build" bisogna selezionare la MainClass per l'esecuzione, che nel nostro caso è la classe PrimaAppWebApplication generata automaticamente da Spring Boot che contiene il codice per far partire l'applicazione web:



Per selezionare la main class basta selezionare le proprietà del progetto e poi nella sezione 'run' cliccare su browser nella riga corrispondente alla proprietà main class



Una volta selezionata la main class si può procedere con una prima esecuzione che mostrerà solo il logo del framework terminando subito.

```
13:28:29.178 [Thread-0] DEBUG org.springframework.boot.devtools.restart.classloader.RestartClassLoader - Created RestartClassLoader org.springframework.boot
:: Spring Boot :: (v2.7.5)
```

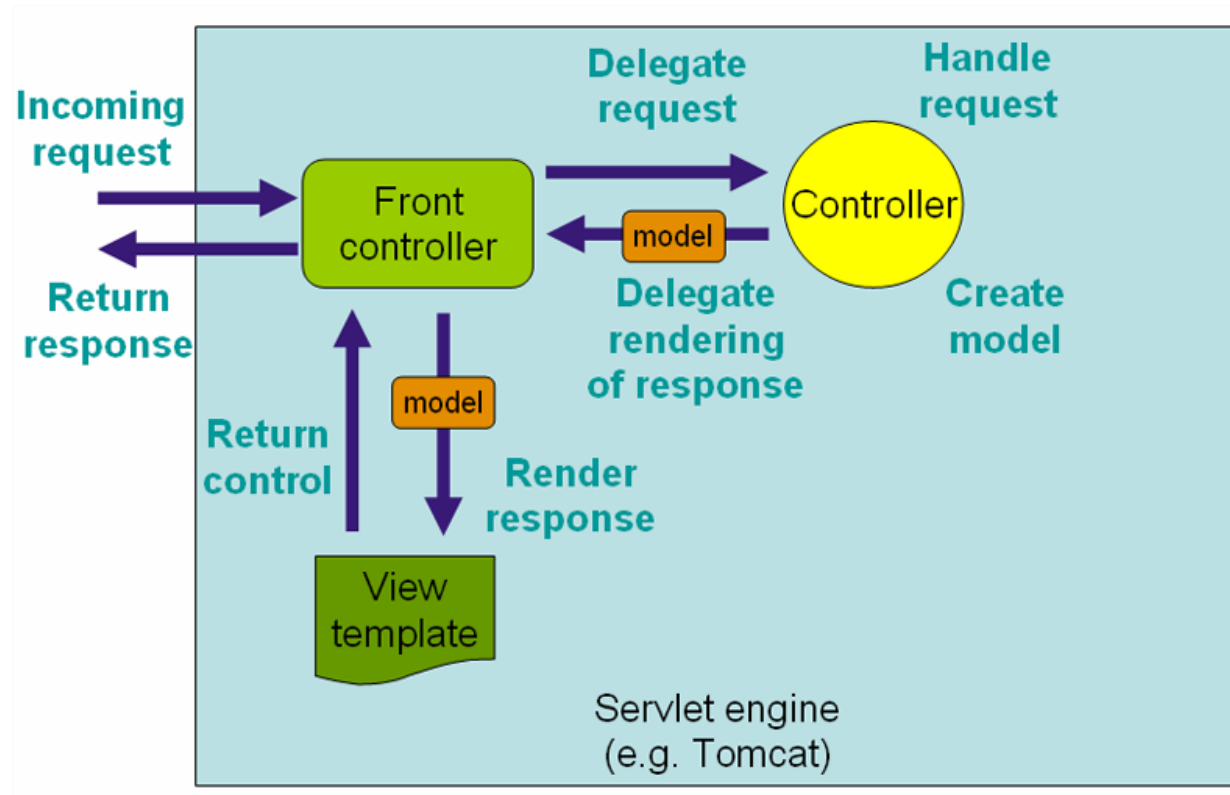
## Esercizio

Creare lo scheletro della prima applicazione web Spring.

## Architettura di un'applicazione web Spring

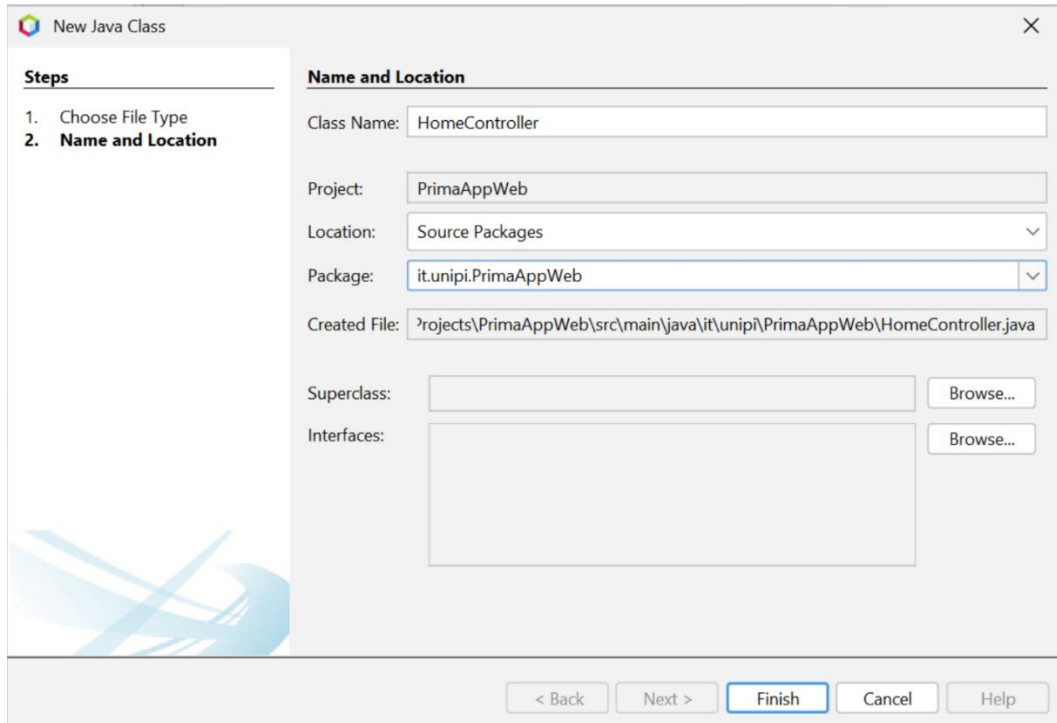
Spring separa la logica applicativa da quella che gestisce la visualizzazione. Una richiesta di una pagina web viene quindi ricevuta da un front controller (nascosto al programmatore) il quale smista la richiesta ad un controller realizzato dal programmatore che riceve e processa la richiesta attraverso una serie di handlers, funzioni associate ad un determinato tipo di richiesta.

Il controller una volta generata la risposta passa la risposta nuovamente al front controller il quale si prenderà carico di effettuare il rendering di come visualizzare i dati di risposta all'utente. La visualizzazione verrà effettuata attraverso un template definito dal programmatore. Con questa struttura il programmatore dovrà definire la sua applicazione web attraverso la logica che genera i dati, e il modo con cui li mostra, garantendo una separazione che permette di gestire al meglio lo sviluppo dell'applicazione.



Aggiunta del Controllore e della prima pagina

Il passaggio successivo è quello di creare un primo controller. Questo può essere fatto definendo una nuova classe:



La classe del controllore deve essere contrassegnata dall'annotazione "@controller" e deve contenere gli handler per la gestione dei tipi diversi di richieste.

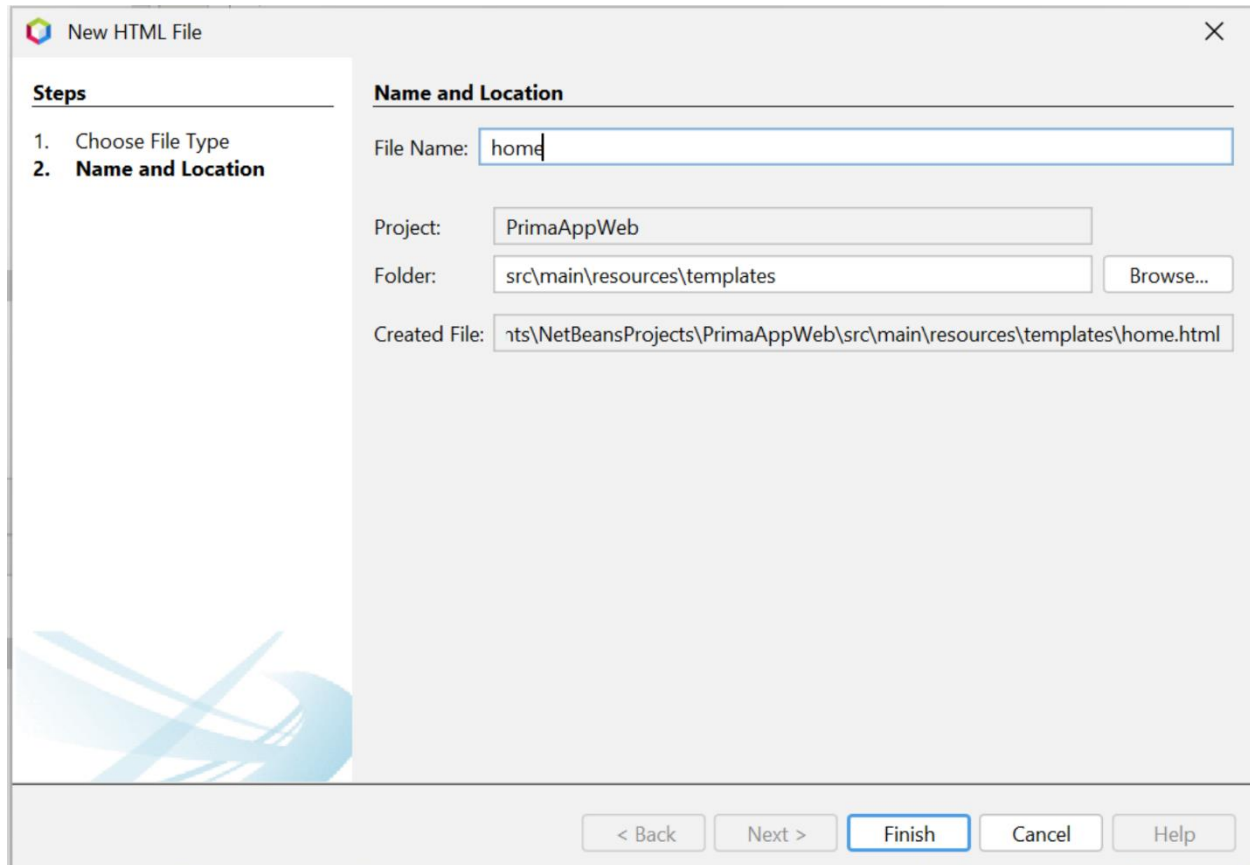
Ogni handler viene contraddistinto dall'annotazione "@GetMapping("PATH")", dove il path specifica l'URL della pagina corrispondente all'handler che deve essere invocato in corrispondenza ad una richiesta GET da parte di un client. Altre annotazioni quali @PostMapping e @PutMapping sono definite per gli altri metodi http.

```
package it.unipi.PrimaAppWeb;

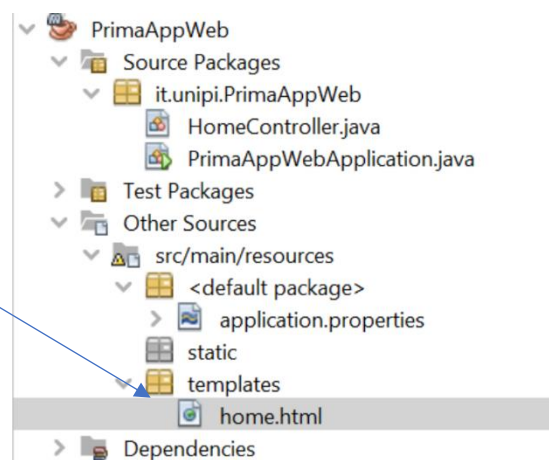
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {
    @GetMapping("/")
    public String home() {
        return "home";
    }
}
```

Ogni handler deve ritornare un riferimento al template di visualizzazione chiamato **view template** dei dati che conterrà il codice HTML per il rendering della pagina. Il prossimo passo è quindi quello di creare un nuovo rendering per la pagina iniziale:



I template vanno memorizzati nella cartella  
src/main/resources/templates

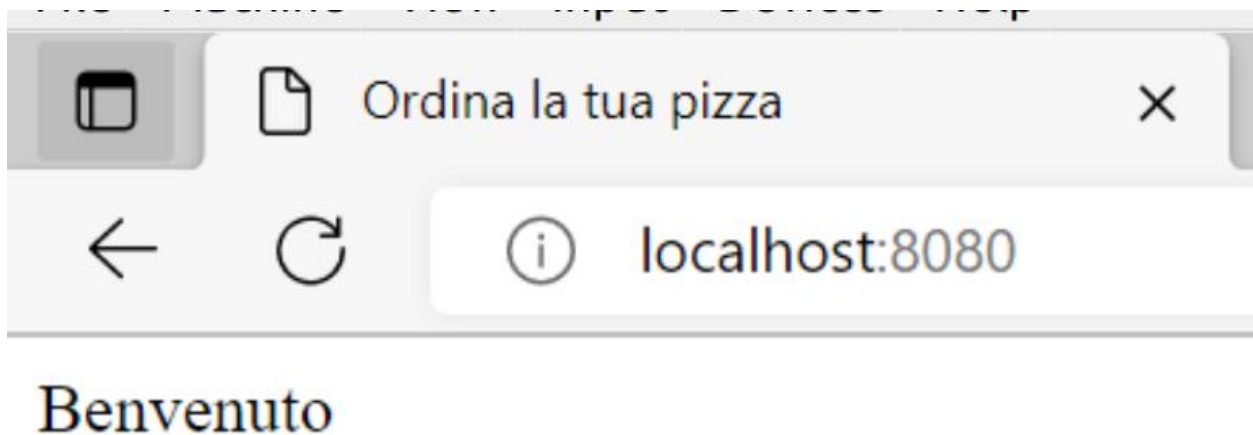


In questa prima pagina di benvenuto includiamo nel template solamente un semplice codice HTML come home page:

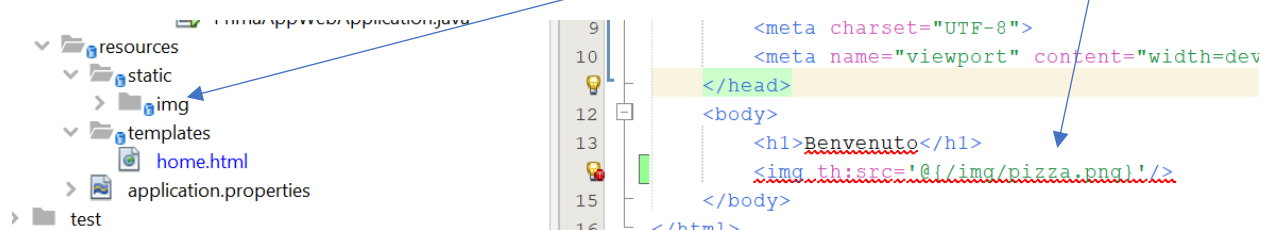
```
<html>
  <head>
    <title>Ordina la tua pizza</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <h1>Benvenuto</h1>
  </body>
</html>
```

Una volta lanciata la nostra applicazione web verrà mostrata tramite l'URL: <http://localhost:8080/>

E il messaggio di benvenuto incluso nel template verrà mostrato:



Proviamo a complicare leggermente la pagina aggiungendo un'immagine. In questo caso l'immagine deve essere memorizzata all'interno del progetto Java nella directory "resources/static/img" che conterrà tutti i contenuti statici dell'applicazione web. Nel nostro template dovremo aggiungere un riferimento:



## Esercizio

Aggiungi una pagina di esempio allo scheletro dell'applicazione web.



## Scambio dati browser / applicazione

Lo scambio dei dati tra il client e l'applicazione avviene attraverso le form HTML, all'interno delle quali gli utenti possono specificare dati da dare in input all'applicazione.

Supponiamo di voler realizzare un sistema per ordini online di Pizze. Come prima cosa abbiamo bisogno di una classe che rappresenti le informazioni associate ad una pizza, in maniera simile a come abbiamo fatto per la serializzazione/deserializzazione degli oggetti in XML/JSON.

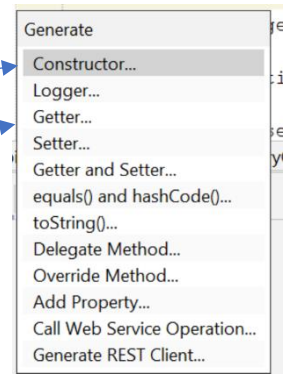
Tali oggetti sono oggetti particolari chiamati JavaBeans che sono costruiti con una determinata convenzione:

```
1. public class Pizza implements Serializable {
2.     private String id;
3.     private String name;
4.     private Type type;
5.
6.     public Pizza() {
7.     }
8.
9.     public Pizza(String id, String name, Type type) {
10.         this.id = id;
11.         this.name = name;
12.         this.type = type;
13.     }
14.
15.     public String getId() {
16.         return id;
17.     }
18.
19.     public String getName() {
20.         return name;
21.     }
22.
23.     public Type getType() {
24.         return type;
25.     }
26.
27.     public void setId(String id) {
28.         this.id = id;
29.     }
30.
31.     public void setName(String name) {
32.         this.name = name;
33.     }
34.
35.     public void setType(Type type) {
36.         this.type = type;
37.     }
38.
39.     public static enum Type { BIANCA, ROSSA, VEGETARIANA };
40. }
41.
```

- Hanno una serie di campi dati, solitamente con un nome che comincia con la lettera minuscola
- Hanno un costruttore senza argomenti
- Hanno un costruttore con tutti i campi che compongono l'oggetto

- Hanno una serie di funzioni per la lettura e modifica dei campi creati con una certa convenzione chiamate funzioni 'getter' e 'setter' rispettivamente

NetBeans aiuta a creare queste classi. Una volta definito il cuore della classe (i campi dati) si può usare la funzione "Insert Code" per includere un costruttore "Constructor" e per includere le funzioni "Setter" e "Getter".



Una volta creata la struttura dati si può andare a programmare l'handler per mostrare a video una serie di pizze e recuperare la scelta dell'utente.

Supponiamo di voler mostrare la lista direttamente nella pagina home e di raccogliere la scelta attraverso una form.

Il codice dell'handler diventa il seguente:

```

1. private static List<Pizza> pizze = Arrays.asList(
2.     new Pizza("MRG", "Margherita", Type.ROSSA),
3.     new Pizza("BNG", "Bianca", Type.BIANCA),
4.     new Pizza("BCP", "Bianca con pomodorini",
5.         Type.BIANCA),
6.     new Pizza("BFL", "Bufala", Type.ROSSA),
7.     new Pizza("VRG", "Verdurosa", Type.VEGETARIANA)
8. );
9. @GetMapping("/")
10. public String home(Model model){
11.
12.     model.addAttribute("tutte", pizze);
13.
14.     return "home";
15. }
16.

```

## Benvenuto



## Scegli la tua pizza

Margherita ▼ Invia pizza scelta

Inizialmente si crea una lista di pizze disponibili (righe 1-7) il quale poi viene passato al view template "home" attraverso la struttura dati Model.

Una volta forniti i dati (l'elenco delle pizze) bisogna modificare il view template per includere una form che mostra l'elenco delle pizze e allo stesso tempo permette di fare una scelta all'utente.

```
1. <h1>Scegli la tua pizza</h1>
2. <form method="POST">
3.   <select id="selezione" name="selezione">
4.     <div th:each="p: ${tutte}">
5.       <option name="pizze" th:value="${p.id}" th:text="${p.name}"></option>
6.     </div>
7.   </select>
8.
9.   <button>Invia pizza scelta</button>
10.
11. </form>
```

Tramite una serie di tag della libreria Thymeleaf<sup>1</sup> di spring si possono specificare delle operazioni sui dati ricevuti. Ad esempio attraverso il tag `th:each="p: ${tutte}"` alla riga 4 si può specificare che il blocco in esso contenuto (il tag `option` nella riga 5) debba essere ripetuto per ogni elemento del vettore 'tutte' passato al template dal controller. Il risultato è che il campo `option` viene ripetuto per ogni pizza, includendo l'ID e il nome attraverso i tag `"th:value"`.

Al momento della pressione del bottone "Invia pizza scelta" il browser effettuerà una POST con l'ID della pizza selezionata nel payload della richiesta.

## Esercizio

Creare la pagina home con la scelta della pizza

---

<sup>1</sup> <https://www.thymeleaf.org/doc/tutorials/2.1/usingthymeleaf.html>

Supponiamo di voler includere nella pagina “/selezione” la funzionalità di inserimento dell’indirizzo di consegna della pizza.

Come nel caso della pizza abbiamo bisogno di creare una struttura dati con i campi dell’indirizzo:

```
1. public class Indirizzo {
2.     private String nome;
3.     private String cognome;
4.
5.     public Indirizzo(String nome, String cognome) {
6.         this.nome = nome;
7.         this.cognome = cognome;
8.     }
9.
10.    public Indirizzo() {
11.    }
12.
13.    public String getNome() {
14.        return nome;
15.    }
16.
17.    public String getCognome() {
18.        return cognome;
19.    }
20.
21.    public void setNome(String nome) {
22.        this.nome = nome;
23.    }
24.
25.    public void setCognome(String cognome) {
26.        this.cognome = cognome;
27.    }
28.
29. }
30.
```

In aggiunta dobbiamo creare un handler con la view template corrispondente.

```
1. @PostMapping("/")
2. public String selezionata(Model model, @RequestParam("selezione") String selezione){
3.     logger.info("Pizza selezionata: "+selezione);
4.
5.     model.addAttribute("indirizzo", new Indirizzo());
6.
7.     return "indirizzo";
8. }
9.
```

In questo caso l’handler avrà un argomento di tipo stringa che conterrà l’ID della pizza selezionata (riga 2). Il campo contenente l’ID della pizza selezionata richiede un’annotazione che ne contraddistingue la corrispondenza, necessaria nel caso di dati semplici. Una nuova istanza della classe Indirizzo sarà passata al view template con riferimento ‘indirizzo’, in questo caso per l’inserimento dei valori dalla form.

In questo caso il view template dovrà contenere una form, in questo caso per collegare il campo della form al campo della struttura dati si usa il tag “th:field”.

```
1. <h1>Fornisci il tuo indirizzo</h1>
2. <form method="POST" th:action="@{/conferma}">
3.
4.     <label for="nome">Nome: </label>
5.     <input type="text" th:field="${indirizzo.nome}"/>
6.     <label for="cognome">Cognome: </label>
7.     <input type="text" th:field="${indirizzo.cognome}"/>
8.
9.     <button>Conferma identità</button>
10. </form>
```

Per gestire la POST effettuata dal browser a seguito dell'inserimento dei dati si dovrà includere un altro handler nel controllore:

```
1. @PostMapping("/conferma")
2. public String conferma(Indirizzo i){
3.     logger.info("Indirizzo: "+i);
4.
5.     return "conferma";
6. }
7.
```

L'istanza della classe Indirizzo contenente i dati riportati dall'utente viene passata come argomento. In questo caso non c'è stato bisogno dell'annotazione dato l'uso della struttura dati e quindi di un campo complesso.

L'ultima pagina mostrata sarà una view "conferma" che può semplicemente contenere un messaggio come il seguente:

```
1. <html>
2.     <head>
3.         <title>Ordina la tua pizza</title>
4.     </head>
5.     <body>
6.         <h1>Benvenuto</h1>
7.         
8.
9.         <h1>Ordine confermato</h1>
10.
11.     </body>
12. </html>
13.
```

## Esercizio

Completare l'applicazione includendo anche una pagina per raccogliere l'indirizzo di consegna e una pagina di conferma.