

**SUPSI**

# Web application frameworks e Spring MVC

# Framework web

- Un **framework per applicazioni web** è un framework software progettato per supportare lo sviluppo di siti web dinamici, **applicazioni web** e **servizi web**.
- Lo scopo del è quello di **alleggerire il lavoro** associato allo sviluppo delle **attività più comuni** di un'applicazione web da parte dello sviluppatore
- Molti framework forniscono ad esempio delle librerie per **l'accesso alle basi di dati**, per la creazione di **template HTML** o per **gestire la sessione** dell'utente
- Lo scopo di un framework è infatti quello di risparmiare allo sviluppatore la riscrittura di codice già scritto in precedenza per compiti simili. L'utilizzo di un framework impone al programmatore una precisa metodologia di sviluppo del software
- [http://en.wikipedia.org/wiki/Comparison\\_of\\_web\\_application\\_frameworks](http://en.wikipedia.org/wiki/Comparison_of_web_application_frameworks)

# Parti in comune di un framework web

- Alcuni parti in comune a più framework web:
  - Central dispatcher
  - Routing
  - View template engine
  - Persistenza dei dati (ORM, Active Records)
  - Sicurezza
  - Scelta librerie javascript
  - Model-View-Controller

# Central dispatcher

- Applicazioni con logica di **navigazione complessa sono più facili da mantenere con un dispatcher (smistatore) centralizzato**
- Molti framework basati sulle servlet e le cui applicazioni, per esempio, possono essere distribuite su Tomcat, usano questo pattern: **hanno una unica servlet che risponde a tutti i percorsi possibili dell'applicazione.**  
In questo caso il routing non è mappato all'interno del web.xml ma con un altro meccanismo (vedi prossima diapositiva)
- Spring MVC per esempio usa una DispatcherServlet

# Routing / mapping

- Per *routing* si intende l'associazione (mappatura) dell'indirizzo (percorso) per una determinata risorsa con la procedura che dà la risposta HTTP
- Per esempio:  
*http://localhost:8080/sample/hello* è mappato alla classe servlet *HelloServlet* nell'applicazione *sample*, in questo caso la mappatura avviene nel *web.xml*
- Ci sono modi di definire il routing, per esempio:
  - In Spring MVC, che utilizza un'unica servlet, la mappatura è fatta all'interno delle classi: tramite annotazioni Java è possibile indicare quale metodo risponde a quale richiesta
  - In .NET MVC, la mappatura è fatta per convenzione: per esempio al percorso */sample/hello* corrisponde una chiamata ad una classe di nome *SampleController* e al metodo *hello*

# Template engine

- 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 file

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

Objects

```
data.name = "Marco"
```

Template  
engine

Output

```
<html>
...
Hello Marco!
...
</html>
```

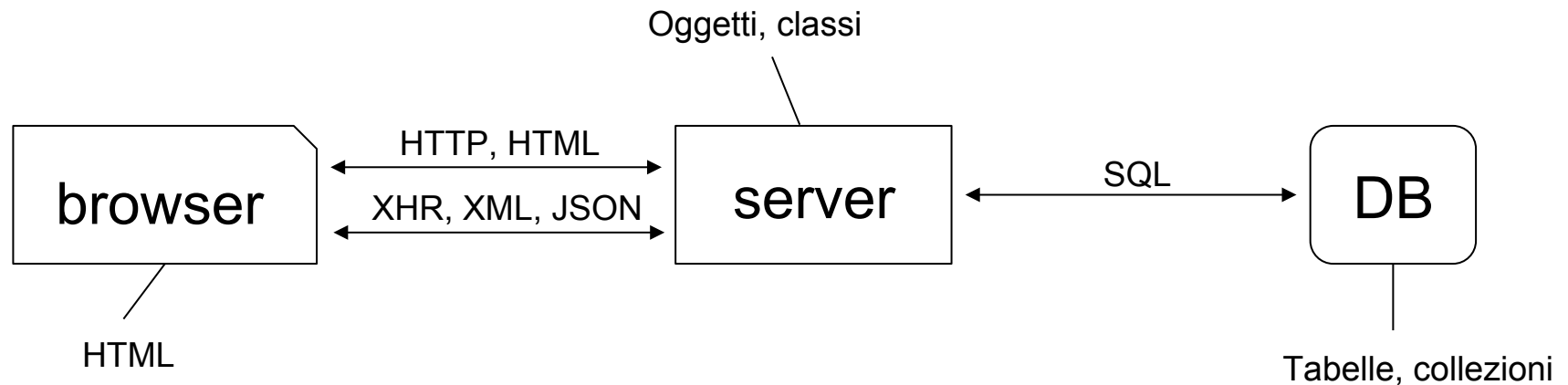
- Alcuni esempi: Razor, Freemarker, Velocity, Jade, Thymeleaf
- [http://en.wikipedia.org/wiki/Comparison\\_of\\_web\\_template\\_engines](http://en.wikipedia.org/wiki/Comparison_of_web_template_engines)

# Persistenza dei dati

- Molti web framework **integrano** al loro interno soluzioni per **la persistenza dei dati**
- Molte soluzioni utilizzano l'**Object-Relational-Mapping** (ORM) per trasformare le tabelle di database relazionali (e non solo, anche nosql) in oggetti utilizzabili all'interno del codice
- Esempi di ORM sono: Entity framework, Hibernate, OpenJPA, TopLink, Doctrine, YII, ...
- **Active Record pattern** è una implementazione di ORM e mappa una riga di una tabella in un oggetto
- **Data Mapper** è un'altra implementazione di ORM

# Rappresentazioni dei dati

- In una applicazione web, i dati (solitamente) sono salvati in un database (quindi in tabelle, collezioni, ... a dipendenza dal tipo di db). Mentre in un application server i dati sono rappresentati da oggetti (e definiti in classi). Nel browser sono visualizzati tramite HTML.
- Nelle richieste ajax i dati vengono (solitamente) rappresentati da documenti XML oppure JSON.





# Esempio *Persona*

```
public class Persona {  
  
    public String nome;  
    public String cognome;  
    public String avs;  
  
}
```

```
CREATE TABLE persona (  
    nome varchar(255),  
    cognome varchar(255),  
    avs varchar(255)  
)
```

# XML

- **XML** (sigla di **eXtensible Markup Language**) è un linguaggio di markup estendibile in quanto permette di creare tag personalizzati.

```
<?xml version="1.0" encoding="UTF-8"?>
<persona>
  <nome>Marco</nome>
  <cognome>Bernasconi</cognome>
  <avs>132.453.32.123</avs>
</persona>
```

# JSON

- **JSON**, acronimo di **JavaScript Object Notation**, è un formato adatto per lo scambio dei dati in applicazioni client-server.

```
{  
  "nome": "Marco",  
  "cognome": "Bernasconi",  
  "avs": "132.453.32.123"  
}
```

# Sicurezza

- La maggior parte dei framework web fornisce un elevato numero di controlli per evitare attacchi tipo:
  - **Sql injection** esecuzione di query arbitrarie sul DB del server
  - **Cross-site scripting (XSS)** accesso ai dati conservati nei cookie o possibili attacchi fishing
  - **Gestione dello stato** accesso a dati riservati
  - ...
- Forniscono metodi per validare i parametri dalle richieste, che potrebbe essere manipolati
- L'OWASP (**Open Web Application Security Project**) è un progetto open-source per la sicurezza delle applicazioni e ha stilato la lista delle top 10 falle di sicurezza delle applicazioni web:  
[https://www.owasp.org/index.php/Top\\_10-2017\\_Top\\_10](https://www.owasp.org/index.php/Top_10-2017_Top_10)

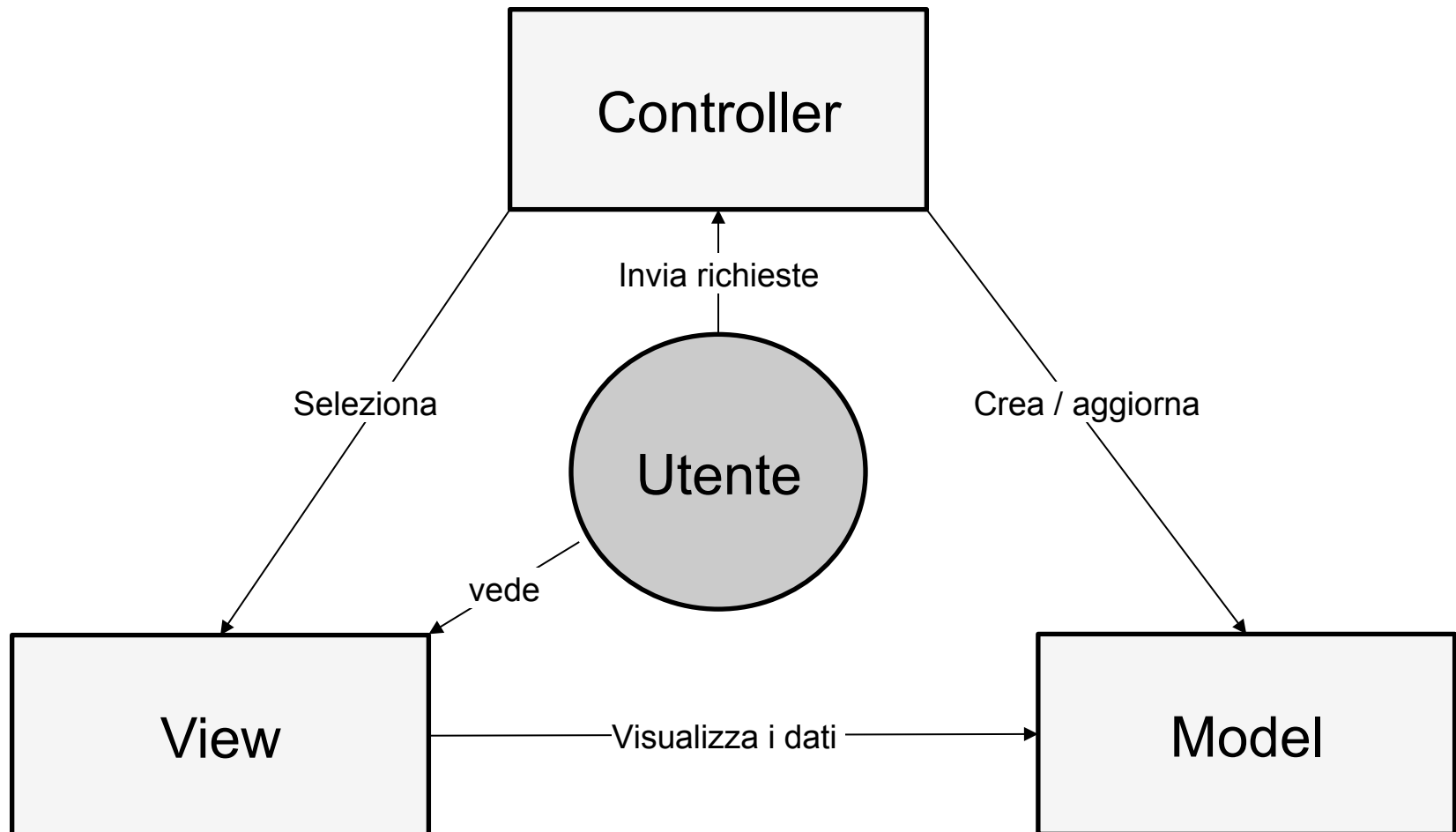
# Scelta librerie javascript

- La maggior parte dei framework integra al proprio interno alcune librerie javascript
- Alcune di queste potrebbero servire soprattutto per
  - manipolazione del documento html
  - le chiamate AJAX (chiamate asincrone via javascript per una pagina ancora più dinamica)
  - la creazione di grafici
  - la gestione delle tabelle (paginazione, filtri, ordine, ...)
- Alcuni esempi: jQuery, Dojo Toolkit, MooTools, Datatables, ...

# Pattern MVC

- Il **Model-View-Controller** è un pattern architetturale in grado di separare la logica di presentazione dei dati dalla logica di business
- È basato sulla separazione dei compiti fra i tre componenti:
  - il **model**: il modello dei dati, le entità e le relazioni tra esse
  - la **view** visualizza i dati contenuti nel model e si occupa dell'interazione con gli utenti (il codice HTML)
  - il **controller** riceve i comandi dell'utente (in genere attraverso il view) e li attua modificando lo stato del modello e chiamando una view
- Molti framework web hanno una architettura MVC

# Pattern MVC nel web



# Alcuni criteri per scegliere un framework

- **Popolarità e grandezza della comunità di programmatori**
  - C'è supporto?
- **Filosofia**
  - Quale tipo di pattern usa? Quanto è personalizzabile?
- **Maturità**
  - Da quanto tempo c'è? È usato in produzione?
- **Documentazione**
  - Scritta bene? Usabile? Tanta?
- **Estensioni e plugin**
  - Quanti plugin ci sono? È facile estenderlo?
- **Sicurezza**
  - Assicura un minimo di sicurezza? È vulnerabile?
- **Linguaggio di programmazione**
- **Licenza**



# Ruby on Rails

- Framework open source
- Scritto in Ruby
- Architettura Model-View-Controller
- I suoi obiettivi sono la semplicità e la possibilità di sviluppare applicazioni con meno codice rispetto ad altri framework. Il tutto con necessità di configurazione minimale.
- Scaffolding: automatizzazione della creazione dei controller e delle view
- I suoi principi guida sono:
  - **Don't repeat yourself**: le definizioni devono essere poste una sola volta
  - **Convention over configuration**: il programmatore deve metter mano alle configurazioni solo per ciò che differisce dalle convenzioni. Ad esempio, se un modello è costituito dalla classe *Post*, la corrispondente tabella nel database deve chiamarsi *posts*, o altrimenti deve essere specificata manualmente
  - **Active record pattern** per la persistenza dei dati

# PHP: Yii, Symfony e Laravel

- Yii framework
  - Model-View-Controller
  - Active Record pattern
  - Autenticazione
  - Scaffolding
- Symfony
  - Model-View-Controller
  - *Twig* come template engine
  - *Doctrine* per la persistenza dei dati
- Laravel
  - Model-View-Controller
  - *Blade* come template engine
  - *Eloquent ORM*: Active Record pattern
  - ...

# Express su Node.js

- Parte server scritta in javascript
- View template engine *Jade*

```
var express = require('express');
var app = express();

app.get('/', function(req, res){
  res.send('Hello World!');
});

var server = app.listen(3000, function() {
  console.log('Listening on port %d', server.address().port);
});
```

# ASP.NET MVC

- Sviluppato da Microsoft, è di tipo Model-View-Controller
- Utilizza *Razor* come view template engine
- Meccanismo di routing globale, che segue una convenzione
- Entity framework (persistenza dati) integrato ottimamente
- Include fin dall'inizio l'autenticazione
- Parte server scritta in C# o VB.NET
- Molto simile a Spring MVC

# Spark framework

- [www.sparkjava.com](http://www.sparkjava.com)
- Piccolo framework ispirato da Sinatra (framework in ruby)
- Nessun XML di configurazione
- Integrazione possibile con vari template engine: Freemarker, Velocity, Mustache
- Manca l'integrazione con la persistenza dei dati
- Questo breve pezzo di codice basta per eseguire l'applicazione

```
import static spark.Spark.*;

public class HelloWorld {
    public static void main(String[] args) {
        get("/hello", (req, res) -> "Hello World");
    }
}
```

# Grails

- Framework open source
- Basato su linguaggio Groovy (linguaggio di scripting per la JVM)
- Riutilizza Hibernate (ORM) e Spring
- View Template engine è GSP (Groovy Server Pages)
- **convention over configuration**
  - Elimina la necessità di aggiungere configurazioni in file XML. Il framework fa invece uso di una serie di regole o convenzioni
- Inspirato da Ruby on Rails

# Spring MVC

- **Spring MVC** è un framework per realizzare applicazioni web basate sul modello MVC sfruttando i punti di forza offerti dal framework Spring come l'**inversion of control** (tramite dependency injection) e la **aspect oriented programming**
- Si occupa principalmente di:
  - mappare metodi e classi Java con determinati url
  - di gestire differenti tipologie di “viste” restituite al client
  - di realizzare applicazioni internazionalizzate (i18n)
- È possibile creare applicazioni distribuibili tramite .war (quindi su Tomcat per esempio) oppure con una versione di Tomcat incorporata

# Spring framework

- È un framework open source per lo sviluppo di applicazioni di tipo Enterprise
- Caratteristiche principali:
  - Dependency Injection
  - Aspect-Oriented Programming including Spring's declarative transaction management
  - Spring MVC web application and RESTful web service framework
  - Foundational support for JDBC, JPA, JMS
  - ...



# Dependency injection

- Dependency injection tradizionale:

```
public class Store {  
    private Item item;  
  
    public Store() {  
        item = new ItemImpl1();  
    }  
}
```

- Usando DI può essere riscritta senza specificare l'implementazione di `Item`

```
public class Store {  
    private Item item;  
    public Store(Item item) {  
        this.item = item;  
    }  
}
```

# Spring Inversion of Control

```
@Configuration
public class AppConfig {
    @Bean
    public Engine engine() {
        return new Engine("v8", 5);
    }
}
```

- `@Configuration` indica che la classe è una sorgente di *Bean definition*.  
L'annotazione `@Bean` indica che l'oggetto ritornato dal metodo deve essere registrato come Bean nel contesto dell'applicazione. Tutti gli oggetti Bean sono di default singleton.

```
@Component
public class Car {

    @Autowired
    public Car(Engine engine) {
        this.engine = engine;
    }
}
```

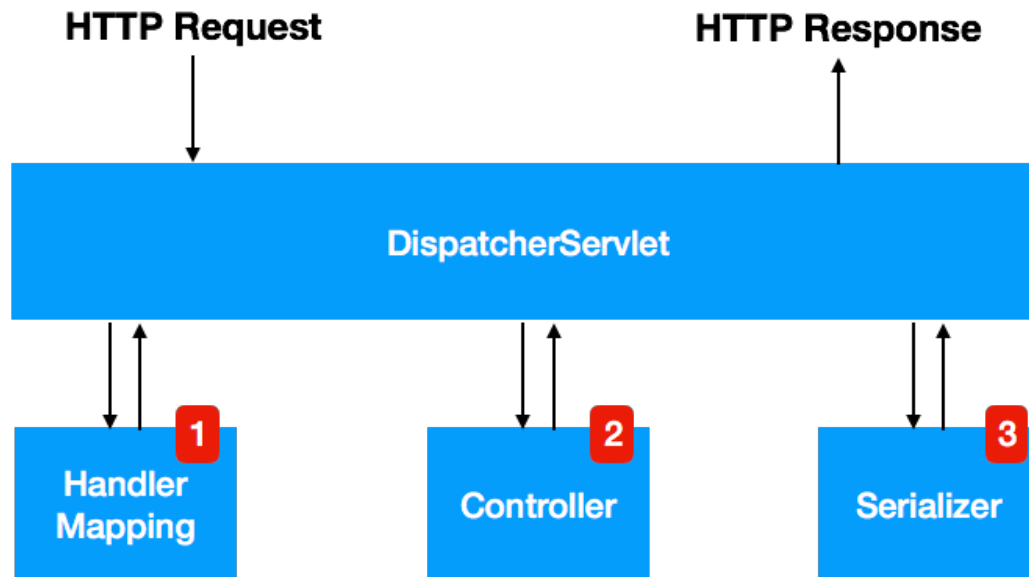
- `@Component` (**e** `@Service` **e** `@Repository`) sono usati per individuare automaticamente i beans usando lo scanning del classpath. L'istanza di `Engine` registrata è iniettata nell'oggetto `Car`.

# Spring MVC

- Implementa perfettamente il pattern MVC:
  - i **Model** sono rappresentati da delle classi che a loro volta rappresentano le entità gestite dall'applicazione
  - le **View** sono rappresentate da vari file che vengono compilati in HTML o da altri tipi di rappresentazioni delle entità (JSON, XML, ...)
  - i **Controller** sono rappresentati da classi (chiamate appositamente 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.
- Per ora utilizzeremo Spring MVC senza ritornare del HTML, ma solo dei documenti JSON
- 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*?
  1. Il *DispatcherServlet* consulta l'oggetto *HandlerMapping* per sapere quale **controller** chiamare (mappatura URL → metodo di una classe)
  2. Il *DispatcherServlet* chiama il metodo corretto del controller, passandogli i parametri richiesti dal metodo. Il metodo esegue quanto richiesto e ritorna un oggetto come risposta
  3. Il *DispatcherServlet*, passa l'oggetto ritornato dal controller ad un serializzatore che lo converte nel formato corretto (JSON, XML, ...)
- Alla fine il *DispatcherServlet* prepara la risposta HTTP con, nel corpo, l'oggetto serializzato

# Spring Boot

- Per creare un'applicazione Spring MVC si utilizza **Spring Boot**, per vari motivi:
  - Permette una partenza molto veloce
  - Usa **convention over configuration** per creare applicazioni Spring con poche, se non zero, configurazioni.
  - Dalle dipendenze dichiarate nel pom.xml capisce quali moduli e caratteristiche avrà l'applicazione
  - Crea applicazioni stand-alone, che si autoconfigurano, con Tomcat incorporato
  - Funziona da interfaccia a linea di comando o da **<http://start.spring.io>**

# http://start.spring.io

## Project Metadata

Artifact coordinates

Group

ch.supsi.dti.webapp

Artifact

blogger

Name

Blogger

Description

Blog platform

Package Name

ch.supsi.dti.webapp.blogger

Packaging

War

Java Version

1.8

Too many options? [Switch back to the simple version.](#)

## Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Web X

DevTools X

Generate Project

# Struttura applicazione Spring MVC

- Una volta generato il progetto, troviamo un progetto maven così strutturato:
- blogger
  - **pom.xml**
  - src
    - main
      - java
        - ch/supsi/dti/webapp/blogger
          - **ServletInitializer.java**
          - **BloggerApplication.java**
    - resources
      - **application.properties**



# Spring boot: pom.xml

- Contiene le dipendenze a:
  - spring-boot-starter-web
  - spring-boot-starter-tomcat
  - spring-boot-starter-test
  - spring-boot-devtools
- Dichiarando queste dipendenze Spring Boot capisce quali moduli e caratteristiche avrà l'applicazione
- Ha già configurato il plugin
  - spring-boot-maven-plugin
- Ha un parent pom
  - spring-boot-starter-parent

# ServletInitializer.java

```
public class ServletInitializer extends SpringBootServletInitializer {  
  
    @Override  
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {  
        return application.sources(BloggerApplication.class);  
    }  
  
}
```

Questa classe consente di utilizzare il supporto alle API Servlet 3.0 di Spring Framework

e consente di configurare l'applicazione quando viene eseguita da un application server come Tomcat.

Senza questa classe e il plugin di Spring Boot in maven, non riusciremmo a pubblicare l'applicazione in Tomcat.

# BloggerApplication.java

```
@SpringBootApplication
public class BloggerApplication {

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

L'annotazione `@SpringBootApplication` è equivalente ad usare le annotazioni `@ComponentScan` e `@EnableAutoConfiguration`

L'annotazione `@ComponentScan` dice a Spring di cercare, ricorsivamente all'interno del suo package e tutti i suoi figli, delle classi marcate direttamente o indirettamente con `@Component`. Per esempio verrà trovato il controller marcato con `@Controller` (è anche una `@Component`)

L'annotazione `@EnableAutoConfiguration` attiva l'auto configurazione dell'applicazione. Viene configurata una `DispatcherServlet` senza il bisogno di avere il `web.xml`.

# Altri modi di eseguire l'applicazione

1. Eseguendo direttamente il metodo main che contiene  
**SpringApplication.run**

Nel nostro esempio eseguendo la classe **BloggerApplication**.

In questo caso Spring boot userà un Tomcat embedded per pubblicare l'applicazione

2. Lanciando il comando **mvn spring-boot:run**
3. Lanciando il comando **mvn package**:
  - poi spostando il war generato in un application server
  - oppure eseguendo **java -jar NOME\_WAR\_GENERATO**

# Spring MVC: application.properties

- **application.properties** può contenere dei parametri di configurazione dell'applicazione
- Inizialmente è vuoto, perchè prende i parametri di default.
- Per esempio è possibile cambiare la porta dove viene lanciato Tomcat:
  - `server.port=8090`
- <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

# Spring MVC: esempio

```
public class Person {  
  
    public String nome;  
    public String cognome;  
    public String avs;  
  
    (...)  
  
}
```

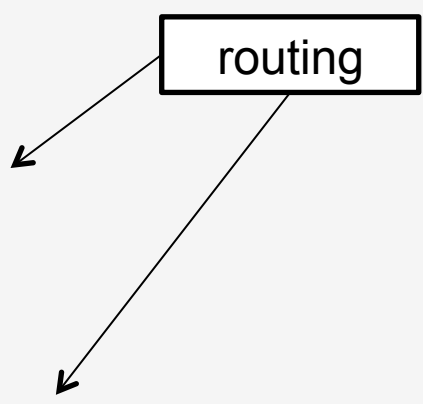
# Spring MVC: esempio

```
@RestController
public class PersonController {

    private List<Person> persons = new ArrayList<>();

    @RequestMapping(value="/person", method=RequestMethod.GET)
    public List<Person> get() {
        return persons;
    }

    @RequestMapping(value="/person", method = RequestMethod.POST)
    public ResponseEntity<Person> post(@RequestBody Person person){
        persons.add(person);
        return new ResponseEntity<Person>(person, HttpStatus.OK);
    }
}
```



The diagram shows a box labeled "routing" with two arrows pointing to the `@RequestMapping` annotations in the code. One arrow points to the GET method's annotation, and the other points to the POST method's annotation.

Nel metodo che gestisce la richiesta in POST, Spring MVC cerca di mappare il JSON ricevuto nel corpo della richiesta in un'oggetto `Person`, questa operazione viene chiamata comunemente *binding*.

La serializzazione in JSON avviene automaticamente, così come la deserializzazione.

# Spring MVC: ResponseEntity

- La classe `ResponseEntity` permette di ritornare l'oggetto da serializzare, ma con la possibilità di poter modificare lo *status code* della risposta, così come tutti gli *header*.

```
@RequestMapping("/handle")
public ResponseEntity<String> handle() {
    URI location = ...;
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.setLocation(location);
    responseHeaders.set("MyResponseHeader", "MyValue");
    return new ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREATED);
}
```

- Potremmo per esempio ritornare `404 NOT FOUND` quando una risorsa richiesta non è stata trovata



# Spring MVC: @RequestMapping

- Con la combinazione di `@RequestMapping` e `@PathVariable` possiamo definire URL dinamici con placeholder:

```
@RequestMapping(value="/person/{id}", method=RequestMethod.GET)
public void getPerson(@PathVariable int id) {
    System.out.println("Requested person " + id);
}
```

- In questo caso abbiamo creato un handler per l'URL `/person/{id}` e, grazie alla nuova annotation `@PathVariable`, abbiamo definito la variabile `id` di tipo `int`. Quindi invocando come url `"/person/10"` verrà invocato il nostro metodo avendo come parametro `"id = 10"`. Ovviamente invocando `/person/aaa` otterremo un errore.

# Spring MVC: esempio

```
>> curl -X POST -d '{"nome":"marco", "cognome":"Bernasconi",  
"avs":"756.1234.5678.97"}' -H "Content-Type: application/  
json" http://localhost:8080/person
```

```
{"nome":"marco", "cognome":"Bernasconi",  
"avs":"756.1234.5678.97"}
```

```
>> curl http://localhost:8080/person
```

```
[{"nome":"marco", "cognome":"Bernasconi",  
"avs":"756.1234.5678.97"}]
```

# Spring MVC: Serializzatore XML

- Di default Spring MVC serializza gli oggetti in JSON.
- Se proviamo a richiedere una risposta in XML (con header nella richiesta `Accept: application/xml`) otterremo una risposta 406 NOT ACCEPTABLE
- È ovviamente possibile cambiare questo comportamento aggiungendo nel `pom.xml` la dipendenza a un serializzatore XML.

Al prossimo riavvio Spring Boot capirà che ora gli è possibile anche serializzare in XML e quindi non ritornerà più 406, ma l'oggetto serializzato in XML.

# Dev tools

- Abitualmente ogni modifica al codice java prevede la compilazione del codice e quindi la ri-esecuzione dell'applicazione. Ciò è molto costoso in termini di tempo (un'applicazione Spring Boot + MVC può metterci da 5 secs ai 20-30 per partire).
- Durante lo sviluppo però è molto scomodo dover aspettare così tanto ad ogni modifica.
- Gli Developer Tools di Spring Boot incudono diversi strumenti per rendere l'esperienza degli sviluppatori migliore. Per esempio: ad ogni modifica del codice, l'applicazione riparte automaticamente e in molto meno tempo.
- <https://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-devtools.html>

# Spring 5 e Spring Boot 2

- <https://docs.spring.io/spring-boot/docs/current/reference/html/>
- <https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html>