

# Corso Web MVC Java EE con Spring

Emanuele Galli

[www.linkedin.com/in/egalli/](http://www.linkedin.com/in/egalli/)

# Design pattern

- È una **soluzione** verificata a un **problema comune**
- Progettazione più flessibile e modulare
- Documentazione del codice più intuitiva
- Testi storici
  - A pattern language – Christopher Alexander – 1977
  - Using Pattern Languages for Object-Oriented Programs – Kent Beck, Ward Cunningham – 1987
  - Design Patterns – Erich Gamma et al. (GoF: Gang of Four) – 1994

# Definizione

- Nome
  - Descrive il pattern e la sua soluzione in un paio di parole
- Problema
  - Contesto e ragioni per applicare il pattern
- Soluzione
  - Elementi del design, relazioni, responsabilità e collaborazioni
- Conseguenze
  - Risultato, costi e benefici, impatto sulla flessibilità, estensibilità, portabilità del sistema
  - Possibili alternative

# Classificazione

- Scopo
  - Creazionali
    - Creazione di oggetti
  - Strutturali
    - Composizione di classi e oggetti
  - Comportamentali
    - Interazione tra oggetti o classi
    - Flusso di controllo
- Raggio d'azione
  - Classe (statico)
    - Ereditarietà
  - Oggetto (dinamico)
    - Associazione
    - Interfacce

# Creazionali

- Singleton (uno e uno solo)
- Factory method (da classi derivate)
- Abstract factory (da famiglie di classi)
- Builder (più step di costruzione)
- Prototype (clone)
- ...

# Strutturali

- Façade (un oggetto rappresenta un sistema)
- Composite (albero di oggetti semplici e composti)
- Decorator (aggiunge metodi dinamicamente)
- Adapter (adatta l'interfaccia a un'altra esigenza)
- Proxy (un oggetto rappresenta un altro oggetto)
- ...

# Comportamentali

- Mediator (interfaccia di comunicazione)
- Observer / Pub-Sub (notifica di cambiamenti di stato)
- Memento (persistenza dello stato)
- Iterator (accesso sequenziale agli elementi)
- Strategy (algoritmo incapsulato in una classe)
- ...

# Altri pattern

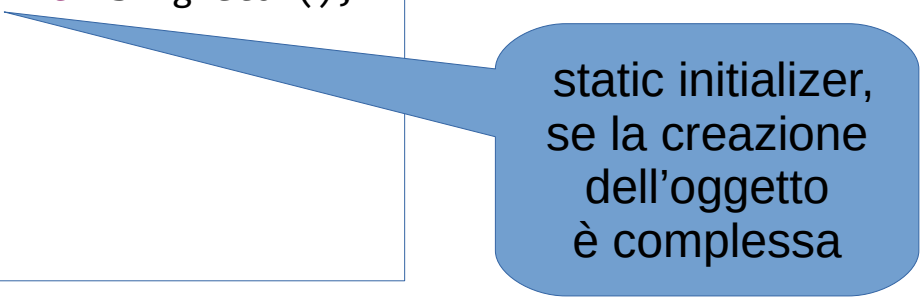
- Funzionali
  - Generator
- Concorrenza
  - Future e promise
- Architetture
  - Model View Controller (MVC)
  - Client/Server



# Singleton

- È necessario che esista un'unica istanza di una classe
- Ctor privato (o protetto), un metodo statico (factory) è responsabile dell'istanziazione e dell'accesso
- Semplice implementazione eager in Java

```
private static final Singleton instance = new Singleton();  
  
private Singleton() {  
}  
  
public static Singleton getInstance() {  
    return instance;  
}
```



static initializer,  
se la creazione  
dell'oggetto  
è complessa

# Singleton (lazy)

```
private static Lazy instance;  
private Lazy() {}  
public static synchronized Lazy getInstance() {  
    if(instance == null) {  
        instance = new Lazy();  
    }  
    return instance;  
}
```

Istanza creata solo su richiesta  
ma la sincronizzazione costa.  
Alternativa: lock tra doppio  
check su instance volatile

Instance creata alla  
prima chiamata

```
private LazyInner() {}  
private static class Helper {  
    private static final LazyInner INSTANCE = new LazyInner();  
}  
public static LazyInner getInstance() {  
    return Helper.INSTANCE;  
}
```

# Strategy

- Modifica dinamicamente un algoritmo
- Il comportamento viene delegato a un'altra classe
- Esempio Java: Comparator per sorting
  - Data una List di Integer
  - Sort con Comparator custom per ordine particolare (prima i numeri dispari e poi quelli pari)

# Strategy con Comparator

```
List<Integer> data = Arrays.asList(42, 7, 5, 12);  
data.sort(new OddFirst());  
System.out.println(data);
```

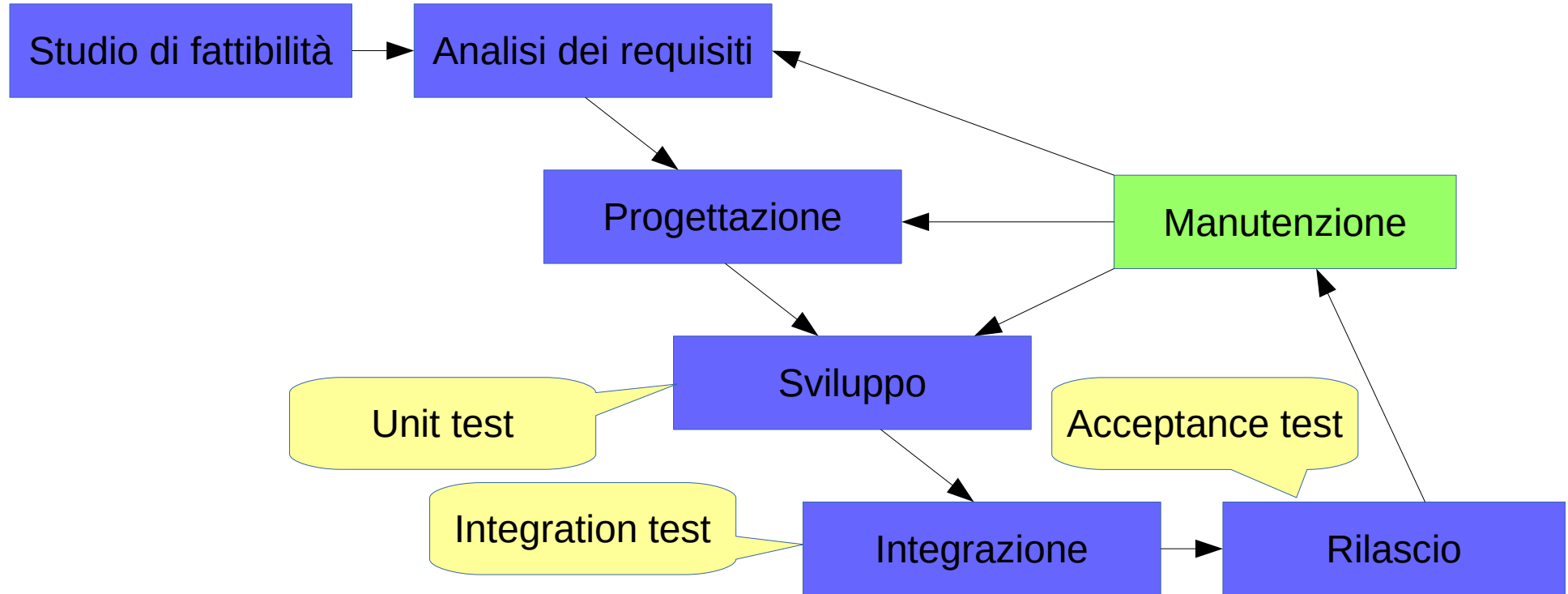
```
class OddFirst implements Comparator<Integer> {  
    @Override  
    public int compare(Integer lhs, Integer rhs) {  
        if (lhs % 2 == 1 && rhs % 2 == 0) {  
            return -1;  
        }  
        if (lhs % 2 == 0 && rhs % 2 == 1) {  
            return 1;  
        }  
        return lhs.compareTo(rhs);  
    }  
}
```

# Ciclo di vita del software

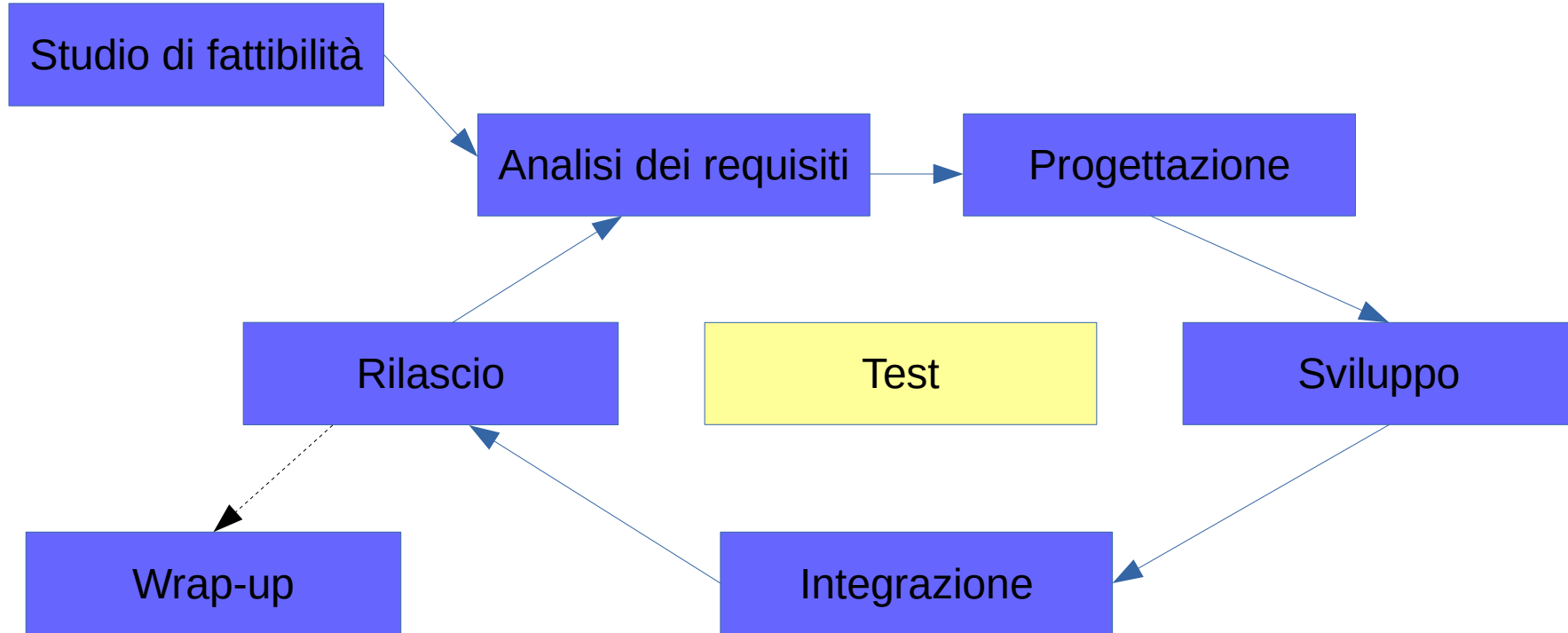
Come gestire la complessità di un progetto?

- Divide et impera
- Struttura
- Documentazione
- Milestones
- Comunicazione e interazione tra partecipanti

# Modello a cascata (waterfall)



# Modello agile



# Build automation con Maven

- Build automation
  - Compilazione del codice sorgente
  - Packaging dell'eseguibile
  - Esecuzione automatica dei test
- UNIX make, Ant, Maven, Gradle
- Apache Maven, supportato da tutti i principali IDE per Java
  - pom.xml (POM: Project Object Model)
  - I processi seguono convenzioni stabilite, solo le eccezioni vanno indicate
  - Le dipendenze implicano il download automatico delle librerie richieste



# Install in Maven

- Da [maven.apache.org](http://maven.apache.org) si può scaricare Maven in formato zip (o tar.gz)
- Basta estrarre l'archivio in una directory dedicata per poter eseguire Maven: “`mvn`” in “`bin`”
- Richiede Java, deve essere definita `JAVA_HOME`
  - Es: `set JAVA_HOME=C:\Program Files\Java\jre1.8.0_211`
- Per verificare che Maven funzioni correttamente: `mvn --version`
- Il repository di Maven viene installato per l'utente corrente in “`.m2`”
- Si può installare un file (jar o altro) nel proprio repository di Maven. Esempio:
  - `mvn install:install-file -Dfile=/app/product/18.0.0/dbhomeXE/jdbc/lib/ojdbc8.jar -DgroupId=com.oracle -DartifactId=jdbc -Dversion=8 -Dpackaging=jar`
  - Il risultato è che `ojdbc8.jar` viene copiato in `.m2\repository\com\oracle\jdbc\8\jdbc-8.jar`

# Nuovo progetto Maven in Eclipse

- Creare un progetto Maven
  - File, New, Project → Wizard “Maven Project”
  - È necessario specificare solo `group id` e `artifact id`
  - Il progetto risultante è per Java 5
- Nel POM specifichiamo le nostre variazioni
  - Properties
  - Dependencies
- A volte occorre forzare l'update del progetto dopo aver cambiato il POM
  - Alt-F5 (o right-click sul nome del progetto → Maven, Update project)

# Properties

- Nel elemento properties del POM si definiscono costanti
- Esempio: quali versioni usare nel progetto per
  - Java (source e target)
  - Il plugin che gestisce i jar in maven

```
<properties>  
  <maven.compiler.source>1.8</maven.compiler.source>  
  <maven.compiler.target>1.8</maven.compiler.target>  
  
  <maven-jar-plugin.version>3.1.1</maven-jar-plugin.version>  
</properties>
```

# Aggiungere una dependency

- Default (“central”) repository:
  - <https://repo.maven.apache.org/maven2>
- Ricerca di dipendenze:
  - <https://search.maven.org/>, <https://mvnrepository.com/>
- Es: JUnit
  - junit → <https://search.maven.org/artifact/junit/junit/4.12/jar>
  - junit jupiter → <https://search.maven.org/artifact/org.junit.jupiter/junit-jupiter/5.5.0/jar>

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
</dependency>
```

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.5.0</version>
</dependency>
```

Tra le <dependencies>

Vogliamo usare Junit  
solo in test,  
perciò aggiungiamo:  
**<scope>test</scope>**

# Import di un progetto mvn in Eclipse

- File, Import ..., Git, Projects from Git
- Clone URI
  - Fornita da GitHub, ad es. <https://github.com/egalli64/swm>
    - Bottone “Clone or download”: <https://github.com/egalli64/swm.git>
- Import as **general project**
- Right click sul progetto
  - Configure, Convert to Maven project

# Spring Framework

- Alternativa allo sviluppo **Java EE** con EJB, usando POJO
- Leggero e modulare, facilmente integrabile con altri framework (Hibernate/JPA)
- **Inversion of Control** (IoC)
  - Hollywood Principle (non chiamare, ti chiamiamo noi)
  - **Dependency Injection** (DI): le proprietà di una classe sono inizializzate da Spring, responsabile per la gestione delle dipendenze
- Aspect Oriented Programming (AOP)
  - Gestione di funzionalità trasversali al progetto

# Riferimenti

- <https://spring.io/>
- Sviluppo
  - <https://spring.io/tools>
    - Spring Tool Suite (**STS**) basata su Eclipse
    - plugin per VS Code e Atom
  - **Eclipse**
    - Marketplace plugin Spring Tools 4 for Spring Boot
  - IntelliJ IDEA
    - Nella Community Edition il supporto è limitato

# Nuovo progetto Spring Boot

- File, New, Spring Starter Project
  - Connessione a <https://start.spring.io>
  - Type Maven, Java Version 8
- Dependencies
  - Web
  - JPA
  - Thymeleaf



# POM

- properties
  - java.version: 1.8
- dependencies
  - spring-boot-starter-web
  - spring-boot-starter-data-jpa (richiede un accesso a database via JDBC)
  - spring-boot-starter-thymeleaf
  - spring-boot-starter-test
  - Oracle JDBC
- build plugin
  - spring-boot-maven-plugin

# Esclusione dipendenze

- Spring Boot Test 2.1.5 usa JUnit 4, ma è pronta per Jupiter.

Esclude  
JUnit 4

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
  <exclusions>
    <exclusion>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Include  
JUnit 5

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <scope>test</scope>
</dependency>
```

Dovrebbe bastare  
La dipendenza  
dall'engine

# application.properties

- In source/main/resources
- Configurazione dell'applicazione

```
spring.jpa.open-in-view=false
```

```
spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
```

```
spring.datasource.url=jdbc:oracle:thin:@localhost:1521/xepdb1
```

```
spring.datasource.username=me
```

```
spring.datasource.password=password
```

```
logging.level.pa=TRACE
```

# Spring Boot Application

- La classe main di una applicazione Spring Boot è annotata `@SpringBootApplication`
- Il suo main chiama `SpringApplication.run()`

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

# Boot Dashboard

- Configurazione del progetto (in local)
  - Right click sul nome del progetto, Open config
  - Spring Boot tab
    - Main type: la nostra SpringBootApplication
- Esecuzione
  - (Re)start o (Re)debug

# Controller

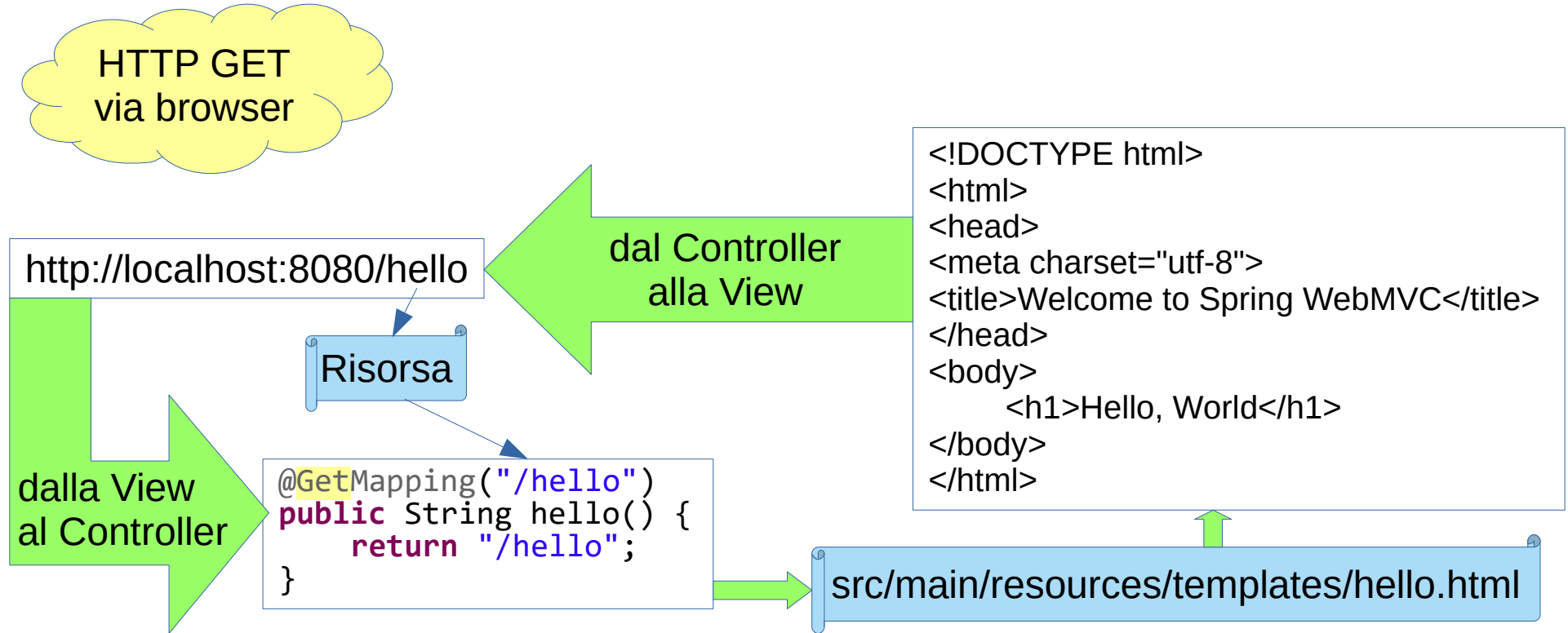
- Ogni classe annotata `@Controller` nel package (o sub) della `@SpringBootApplication` fa parte per default del controller dell'applicazione

```
package dd;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;

@Controller
public class PaController {
    private static final Logger logger = LoggerFactory.getLogger(PaController.class);
    // ...
}
```

# Request / Response



# Thymeleaf

- Template engine in Java
  - <https://www.thymeleaf.org/>
- Template “naturale” → visualizzabile dal browser
- Documentazione
  - <https://www.thymeleaf.org/documentation.html>



# GET e POST

- GET e POST sono i due comandi HTTP comunemente gestiti
- Si annota il metodo come
  - @GetMapping o @PostMapping
- I parametri della request sono mappati nei parametri del metodo via
  - @RequestParam
- La parte di logica Java del controller e quella di generazione dell'HTML (via Thymeleaf, nel nostro caso) comunicano per mezzo di un oggetto `org.springframework.ui.Model`

# Spring Model

- L'interfaccia Model definisce l'accesso a un gestore di attributi in forma chiave-valore.
- La parte Java del controller aggiunge elementi
  - `Model.addAttribute(String name, Object value);`
- Thymeleaf accede agli attributi in lettura

# Controller: Java → Thymeleaf

```
@GetMapping("/guest")
public String guest(@RequestParam(name = "user") String user, Model model) {
    model.addAttribute("user", user);
    return "/guest";
}
```

Definizione del namespace th  
Evita i warning per gli attributi th

Attributo Thymeleaf text

```
<!DOCTYPE html>
<!-- guest.html -->
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8">
<title>Welcome to Thymeleaf</title>
</head>
<body>
<h1 th:text="'Hello, ' + ${user} + '!'"></h1>
</body>
</html>
```

# Object Relational Mapping (ORM)

- Mappaggio della struttura relazionale propria di un database SQL con l'approccio ad oggetti di un linguaggio come Java
- Supporta un design modulare dell'applicazione
- La programmazione in Java risulta più naturale e snella
- Il codice più comunemente usato (**CRUD**: Create, Retrieve, Update, Delete) può essere disponibile by default
- La dipendenza nei confronti del database è più ridotta

# JPA – Hibernate – Spring Data

- Java Persistence API
  - Descrive la gestione di dati relazionali in Java
- Hibernate (e altri ORM per Java) sono stati riscritti per implementare JPA
  - <http://hibernate.org/>
- Spring Data supporta JPA via Hibernate (default)
  - <https://spring.io/projects/spring-data-jpa>

# Entity

- **@Entity**  
Java Bean che rappresenta un oggetto mappato nel database  
JPA assume una relazione con tabella omonima
- **@Table**  
nome della tabella da associare all'entity
- **@Id**  
specifica l'identificatore univoco dell'oggetto
- **@Column**  
mappa una proprietà della classe a una colonna della tabella

```
@Entity
@Table(name = "REGIONS")
public class Region {
    @Id
    @Column(name = "REGION_ID")
    private long id;

    @Column(name = "REGION_NAME")
    private String name;

    // etc.
```

# Repository

- @Repository
  - Interfaccia che definisce funzionalità di accesso al database  
Il codice del repository viene generato dal provider JPA
  - CrudRepository implementa funzionalità CRUD di base, richiede:
    - Il tipo dell'entity che intendiamo utilizzare
    - Il tipo dell'identificatore univoco dell'entity

```
@Repository  
public interface RegionRepository extends CrudRepository<Region, Long> {  
}
```

# Uso di un repository

- Dependency Injection

Variabile di istanza `@Autowired` nel controller

- Nei metodi `@XxxMapping` del controller si richiamano i metodi del repository e si passano i risultati a Thymeleaf

```
@Controller public class RegionCtrl {  
    @Autowired RegionRepo repo;  
  
    @GetMapping("/regions")  
    public String regions(Model model) {  
        model.addAttribute("regions", repo.findAll());  
        return "/regions";  
    }  
}
```



# Metodi in CrudRepository

- count()
- delete(T)
- deleteAll()
- deleteAll(Iterable<>)
- deleteById(ID)
- existsById(ID)
- findAll()
- findAllById(Iterable<>)
- findById(ID)
- save(S)
- saveAll(Iterable<S>)

# findById()

- Ricerca di una Entity per id
- Ritorna un **Optional** (Java 8)
  - isPresent()
  - get()
  - orElse()
  - ...

```
Optional<Region> region = repo.findById(id);  
if (region.isPresent()) {  
    Region cur = region.get();  
  
    // ...  
} else {  
    // ...  
}
```

# Visualizzare una lista con Thymeleaf

```
<p>
  Regions as string: <span th:text="${regions}">[]</span>
</p>

<p>Regions in table</p>
<table>
  <tr>
    <th>ID</th>
    <th>NAME</th>
  </tr>
  <tr th:each="cur: ${regions}">
    <td th:text="${cur.id}">1</td>
    <td th:text="${cur.name}">Europe</td>
  </tr>
</table>
```

toString()

Loop for each

Getter dedotti dai nomi delle proprietà

# Many to One: JPA

- Proprietà che rappresenta una riga della tabella referenziata
- Annotata
  - ManyToOne
  - JoinColumn
    - name
    - referencedColumnName

```
@Entity
@Table(name = "COUNTRIES")
public class Country {
    @Id
    @Column(name = "COUNTRY_ID")
    private String id;

    @Column(name = "COUNTRY_NAME")
    private String name;

    @ManyToOne
    @JoinColumn(name = "REGION_ID")
    private Region region;

    // etc.
```

# Many to One: Thymeleaf

```
<table>
  <tr>
    <th>ID</th>
    <th>NAME</th>
    <th>REGION</th>
  </tr>
  <tr th:each="cur: ${countries}">
    <td th:text="${cur.id}">IT</td>
    <td th:text="${cur.name}">Italy</td>
    <td th:text="${cur.region.name}">Europe</td>
  </tr>
</table>
```

Loop for each

# One To Many: JPA

```
@Entity
@Table(name = "REGIONS")
public class Region {
    @Id
    @Column(name = "REGION_ID")
    private long id;

    @Column(name = "REGION_NAME")
    private String name;

    @OneToMany(mappedBy = "region", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Set<Country> countries;

    // etc.
```

# One to Many: Thymeleaf

for each region

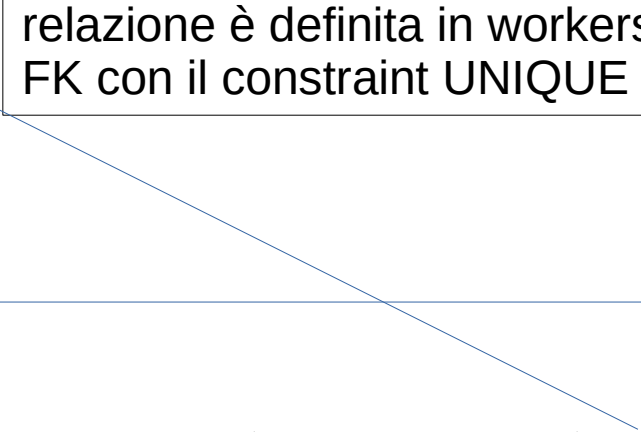
```
<div th:each="region: ${regions}">
  <p>
    <span th:text="${region.id}">1</span>:
    <span th:text="${region.name}">Europe</span>
  </p>
  <table>
    <tr>
      <th>ID</th>
      <th>NAME</th>
    </tr>
    <tr th:each="cur: ${region.countries}">
      <td th:text="${cur.id}">IT</td>
      <td th:text="${cur.name}">Italy</td>
    </tr>
  </table>
</div>
```

for each country  
in this region

# One to One: Database

```
create table workstations(  
    workstation_id integer primary key,  
    name varchar2(20)  
);  
  
insert into workstations values(1, 'deepblue');  
insert into workstations values(2, 'wotan');
```

In questa implementazione, la relazione è definita in workers via FK con il constraint UNIQUE



```
create table workers(  
    worker_id integer primary key,  
    name varchar2(20),  
    workstation_id integer referencing workstations(workstation_id) unique  
);  
  
insert into workers values(10, 'ken thompson', 2);  
insert into workers values(11, 'dennis ritchie', 1);
```



# One to One: JPA

```
@Entity
@Table(name = "WORKSTATIONS")
public class Workstation {
    @Id
    @Column(name = "WORKSTATION_ID")
    private long id;

    @Column(name = "NAME")
    private String name;

    @OneToOne(mappedBy = "workstation")
    private Worker worker;

    // etc.
```

Per usare la relazione, Workstation fa riferimento alla sua definizione in Work

NB: la relazione è definita solo in Worker

```
@Entity
@Table(name = "WORKERS")
public class Worker {
    @Id
    @Column(name = "WORKER_ID")
    private long id;

    @Column
    private String name;

    @OneToOne
    @JoinColumn(name = "WORKSTATION_ID")
    private Workstation workstation;

    // etc.
```

# One to One: Thymeleaf

```
<table>
  <tr><th>ID</th><th>NAME</th><th>WORKSTATION</th></tr>
  <tr th:each="cur: ${workers}">
    <td th:text="${cur.id}">1</td>
    <td th:text="${cur.name}">Guido van Rossum</td>
    <td th:text="${cur.workstation.name}">Spam</td>
  </tr>
</table>
```

```
<table>
  <tr><th>ID</th><th>NAME</th><th>WORKER</th></tr>
  <tr th:each="cur: ${workstations}">
    <td th:text="${cur.id}">1</td>
    <td th:text="${cur.name}">Spam</td>
    <td th:text="${cur.worker.name}">Guido van Rossum</td>
  </tr>
</table>
```

# Many to Many: Database

```
create table participants(  
  participant_id integer primary key,  
  name varchar2(20)  
);  
  
insert into participants values(10, 'ken');  
insert into participants values(11, 'dennis');  
insert into participants values(12, 'bjarne');
```

```
create table projects(  
  project_id integer primary key,  
  name varchar2(20)  
);  
  
insert into projects values(1, 'fuchsia');  
insert into projects values(2, 'wotan');
```

```
create table participant_project(  
  participant_id integer references participants(participant_id),  
  project_id integer references projects(project_id),  
  primary key(participant_id, project_id)  
);  
  
insert into participant_project values(10, 1);  
insert into participant_project values(10, 2);  
insert into participant_project values(11, 1);  
insert into participant_project values(12, 1);  
insert into participant_project values(12, 2);
```

# Many to Many: JPA

```
@Entity
@Table(name = "PARTICIPANTS")
public class Participant {
    @Id
    @Column(name = "PARTICIPANT_ID")
    private long id;

    @Column
    private String name;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "PARTICIPANT_PROJECT", //
        joinColumns = @JoinColumn(name = "PARTICIPANT_ID"), //
        inverseJoinColumns = @JoinColumn(name = "PROJECT_ID"))
    Set<Project> projects;

    // etc.
```

Le definizioni delle due entità  
sono simmetriche

# Many to Many: Thymeleaf

for each project

for each participant  
in this project

```
<h1>Projects</h1>
<div th:each="project: ${projects}">
  <h2 th:text="${project.name}">Wotan</h2>

  <div th:each="cur: ${project.participants}">
    <p th:text="|${cur.id}: ${cur.name}|">1: Tom Smith</p>
  </div>
</div>
```

"|\${cur.id}: \${cur.name}|"  
==  
"\${cur.id} + ': ' + \${cur.name}"

# Sequenza JPA per Oracle

```
@Entity
@Table(name = "EMPLOYEES")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "EmpGen")
    @SequenceGenerator(sequenceName = "EMPLOYEES_SEQ", allocationSize = 1, name = "EmpGen")
    @Column(name = "EMPLOYEE_ID")
    private long id;

    // ...
}
```

valore generato da JPA  
strategy: sequenza fornita dal database  
generator: nome JPA

informazioni sul generatore  
sequenceName: nome ORACLE  
allocationSize: INCREMENT della sequenza  
name: nome JPA

# JPA query

- Nel nostro repository JPA possiamo dichiarare metodi custom “findByXxx()” per operare su proprietà diverse da ID
- Devono ritornare un **Iterable** (o classe derivata)
- In caso di **relazione**, è possibile navigare nell’entità (via “**\_**”)

```
@Repository
public interface CountryRepository extends CrudRepository<Country, String> {
    Iterable<Country> findByName(String name);
    Iterable<Country> findByRegion(Region region);
    Iterable<Country> findByRegion_id(long id);

    // ...
}
```

# Altri esempi per Thymeleaf

```
<link rel="stylesheet" type="text/css" th:href="@{/css/myStyles.css}">
```

```
<select name="id">  
  <option th:each="cur: ${employees}" th:value="${cur.id}"  
    th:text="${cur.id}">...</option>  
</select>
```

```
<h1>Countries<span th:text=" ${message}" th:if="${message}"></span></h1>
```

```
<div th:unless="${region.name.isEmpty()}">  
  <!-- -->  
</div>
```



# Altri esempi Thymeleaf /2

```
<select>
  <option th:each="cur: ${regions}" th:value="${cur.id}" th:text="${cur.name}"
    th:selected="${cur.id==defaultRegion}"></option>
</select>
```

```
<div th:object="${data[0]}">
  <div>Id: <span th:text="*{id}">...</span>.</div>
  <div>Name: <span th:text="*{name}">...</span>.</div>
  <div>Id converted to roman:
    <th:block th:switch="*{id}">
      <span th:case="1"> I </span>
      <span th:case="2"> II </span>
      <span th:case="*"> III </span>
    </th:block>
  </div>
</div>
```