**SUPSI**

# Spring Data

# Injection con @Autowired

- L'annotazione @Autowired permette a Spring di iniettare oggetti all'interno di altri.

```
@Component
public class PersonService {
    public List<Person> getAll() {
        return persons;
    }
}

@Controller
public class PersonController {
    @Autowired
    private PersonService personService;

}
```

# Spring e dependency injection

- Una classe annotata con `@Component` è auto-scoperta da Spring utilizzando lo scanning del classpath.
- Spring quindi creerà una nuova instanza di questa classe, che potrà poi essere utilizzata da altre classe.
- `@Service`, `@Repository`, `@Controller` sono tutte specilizzazioni di `@Component`
- `@RestController` è una shortcut per `@Controller` + `@ResponseBody`
- Quando Spring crea una nuova istanza è di default singleton. Ma non è l'unica possibilità (ci sono anche request, session, …).

# @RestController vs @Controller

```
@Controller
public class PersonController {
    @GetMapping("/")
    @ResponseBody
    public List<BlogPost> list(){
        return posts;
    }
}
```

È uguale a scrivere:

```
@RestController
public class PersonController {
    @GetMapping("/")
    public List<BlogPost> list(){
        return posts;
    }
}
```

# Spring Data

- Spring Data's mission is to provide a **familiar and consistent**, **Spring-based programming model for data access** while still retaining the special traits of the underlying data store.

- It makes it **easy to use** data access technologies**, relational and non-relational databases, map-reduce frameworks, and cloud-based data services**. This is an umbrella project which **contains many subprojects** that are specific to a given database. The projects are developed by working together with many of the companies and developers that are behind these exciting technologies.

- Da http://spring.io/projects/spring-data

# Spring Data modules

- Contiene diversi moduli:
  - Spring Data Commons
  - **Spring Data JPA**
  - Spring Data Key-Value
  - Spring Data MongoDB
  - Spring Data Redis
  - Spring Data for Apache Cassandra
  - Spring Data for Apache Solr
  - ….e molti altri ancora
- In pratica per ogni tipo di tecnologia di persistenza esiste un sottoprogetto Spring Data adatto.

# Esempio *Persona*

```
public class Persona {
    pulic int id;
    public String nome;
    public String cognome;
    public String avs;

}
```

```
CREATE TABLE persona (
    id int(11),
    nome varchar(255),
    cognome varchar(255),
    avs varchar(255)
)
```

# Repository

- L'elemento principale con il quale si lavora utlizzando Spring Data è il **Repository**.
- Per creare un nuovo Repository è sufficiente creare un interfaccia che estende `CrudRepository`.
- `CrudRepository` fornisce funzionalità CRUD (Create, Read, Update, Delete) sofisticate per l'entità che gestisce.

```
@Repository
public interface PersonRepository extends CrudRepository<Persona, Integer> {}
```

- Prende l'entità da gestire e il tipo della sua chiave primaria come type arguments.

# CrudRepository

- L'interfaccia CrudRepository espone questi metodi:

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity);      (1)

    T findOne(ID primaryKey);            (2)

    Iterable<T> findAll();               (3)

    Long count();                        (4)

    void delete(T entity);               (5)

    boolean exists(ID primaryKey);       (6)

    // … more functionality omitted.
}
```

(1) Saves the given entity.
(2) Returns the entity identified by the given id.
(3) Returns all entities.
(4) Returns the number of entities.
(5) Deletes the given entity.
(6) Indicates whether an entity with the given id exists.

# Query creation

- Oltre alle operazioni CRUD si possono facilmente aggiungere queries più specifiche

```
public interface PersonRepository extends Repository<User, Long> {

  List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

  // Enables the distinct flag for the query
  List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
  List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

  // Enabling ignoring case for an individual property
  List<Person> findByLastnameIgnoreCase(String lastname);
  // Enabling ignoring case for all suitable properties
  List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

  // Enabling static ORDER BY for a query
  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

# Query creation

| Keyword | Sample | JPQL snippet |
|---|---|---|
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Is,Equals | findByFirstname, findByFirstnameIs, findByFirstnameEquals | … where x.firstname = ?1 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age <= ?1 |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where x.age >= ?1 |
| After | findByStartDateAfter | … where x.startDate > ?1 |
| Before | findByStartDateBefore | … where x.startDate < ?1 |
| IsNull | findByAgeIsNull | … where x.age is null |
| IsNotNull,NotNull | findByAge(Is)NotNull | … where x.age not null |
| Like | findByFirstnameLike | … where x.firstname like ?1 |

# Query creation

| | | |
|---|---|---|
| NotLike | findByFirstnameNotLike | … where x.firstname not like ?1 |
| StartingWith | findByFirstnameStartingWith | … where x.firstname like ?1 (parameter bound with appended %) |
| EndingWith | findByFirstnameEndingWith | … where x.firstname like ?1 (parameter bound with prepended %) |
| Containing | findByFirstnameContaining | … where x.firstname like ?1 (parameter bound wrapped in %) |
| OrderBy | findByAgeOrderByLastnameDesc | … where x.age = ?1 order by x.lastname desc |
| Not | findByLastnameNot | … where x.lastname <> ?1 |
| In | findByAgeIn(Collection<Age> ages) | … where x.age in ?1 |
| NotIn | findByAgeNotIn(Collection<Age> ages) | … where x.age not in ?1 |
| True | findByActiveTrue() | … where x.active = true |
| False | findByActiveFalse() | … where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | … where UPPER(x.firstame) = UPPER(?1) |

# @Query

- Con l'annotazione `@Query` è possibile aggiungere delle query JPA:

```
@Repository
public interface BlogPostRepository extends JpaRepository<BlogPost, Integer> {

   // con query creation
   List<BlogPost> findOneByCategory(String category);

   // query JPA con annotazione
   @Query("SELECT b FROM BlogPost b where b.category = :value")
   List<BlogPost> list(@Param(value = "value") String category);
}
```

# Accesso DB con Spring MVC e JPA

- Grazie a Spring Data JPA è facile integrare l'accesso ad un database in una applicazione Spring MVC
- È sufficiente:
  - aggiungere la dipendenza a *spring-boot-starter-data-jpa* nel progetto
  - annotare le classi che corrispondono a delle tabelle nel db con la sintassi **JPA**
  - creare una classe che estende `JpaRepository` e annotarla `@Repository`
  - Per esempio:

```
@Repository
public interface PersonRepository extends JpaRepository<Persona, Integer>
{
   public Iterable<Persona> findPersonsByName(String nome);
}
```

- Da notare che `PersonRepository` è una interfaccia. `JpaRepository` è una sottoclasse di `CrudRepository`.

# Esempio *Persona*

```
@Entity
public class Persona {

    @Id
    @GeneratedValue
    public int id;

    public String nome;
    public String cognome;
    public String avs;

}
```

# application.properties

- Per configurare la connessione al database e il funzionamento di Hibernate servono almeno queste proprietà:

```
spring.datasource.url=jdbc:mysql://localhost:3306/db
spring.datasource.username=marco
spring.datasource.password=123465

spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
```

- In questo caso, con **update** indichiamo ad hibernate di creare le tabelle nel database automaticamente e di modificarle se le classi dovrebbero cambiare.

# Spring Data JPA

- È possible configurare Spring Data JPA in modo che lavori con database in modalità client-server (mysql, oracle, …), ma anche con database in memoria
- Aggiungendo alle dipendenze del progetto la libreria del database, questo sarà automaticamente riconosciuto
- È possibile, per esempio, far partire automaticamente un database HyperSQL in memoria solo aggiungendo la dipendenza nel file pom.xml (di maven)
- Oppure aggiungere la dipendenza al connettore JDBC di MySQL, per lavorare con MySQL

# data.sql

- Spring Boot consente di default di caricare degli script sql

- inserendo alla radice del classpath un file ***data.sql***. Quando l'applicazione si avvia lo script al suo interno verrà eseguito

# Link utili

- http://spring.io/projects/spring-data
- https://docs.spring.io/spring-data/jpa/docs/current/reference/html/
- http://spring.io/projects/spring-data-jpa
- https://spring.io/guides/gs/accessing-data-jpa/