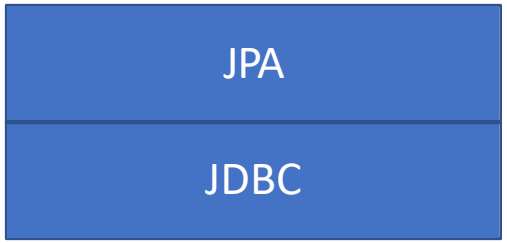


JAVA EE

JAVA SE

JPA = JAVA PERSISTENCE API = INSIEME DI API RILASCIATE DA JAVA EE CHE CONSENTONO L'IMPLEMENTAZIONE DI OPERAZIONI DI CRUD SU UN DATABASE RELAZIONALE, RILASCIATE SOTTO FORMA DI ORM (OBJECT RELATIONAL MAPPING)

UN ORM E' UN INSIEME DI API CHE FORNISCONO REGOLE PRECISE PER COSTRUIRE UNO SPECCHIO APPLICATIVO DELLA STRUTTURA FISICA DEL DATABASE RELAZIONALE

IN PARTICOLARE JPA STABILISCE DELLE REGOLE SECONDO LE QUALI PER OGNI TABELLA DI UN DATABASE RELAZIONALE OCCORRE CREARE UNA CLASSE SECONDO DELLE SPECIFICHE REGOLE

REGOLE DI MAPPING DI UNA ENTITY JPA RISPETTO AD UNA TABELLA STAND ALONE DI UN DATABASE RELAZIONALE

1. Ogni Classe corrispondente ad una tabella deve essere annotata con l'annotation @Entity

2. Ogni Classe corrispondente ad una tabella deve contenere tante variabili di istanza per quante sono le colonne della corrispondente tabella, tanti metodi di set e di get per quante sono le variabili di istanza, e almeno il costruttore di default

3. Ogni variabile di istanza che rappresenta una PRIMARY KEY di una tabella deve essere annotata con l'annotation @Id

4.Ogni variabile di istanza che rappresenta un AUTO INCREMENT di una tabella deve essere annotata con l'annotation @GeneratedValue(strategy=GenerationType.IDENTITY)

5.Ogni Classe corrispondente ad una tabella deve implementare la marker interface Serializable

JPA NECESSITA DI UN PERSISTENCE PROVIDER

UN PERSISTENCE PROVIDER E' UN PROCESSO A RUNTIME CHE E' IN GRADO DI TRADURRE OPERAZIONI JPA SCRITTE CON LINGUAGGIO JAVA IN CODICE SQL

JPA può essere utilizzato in due modalità :

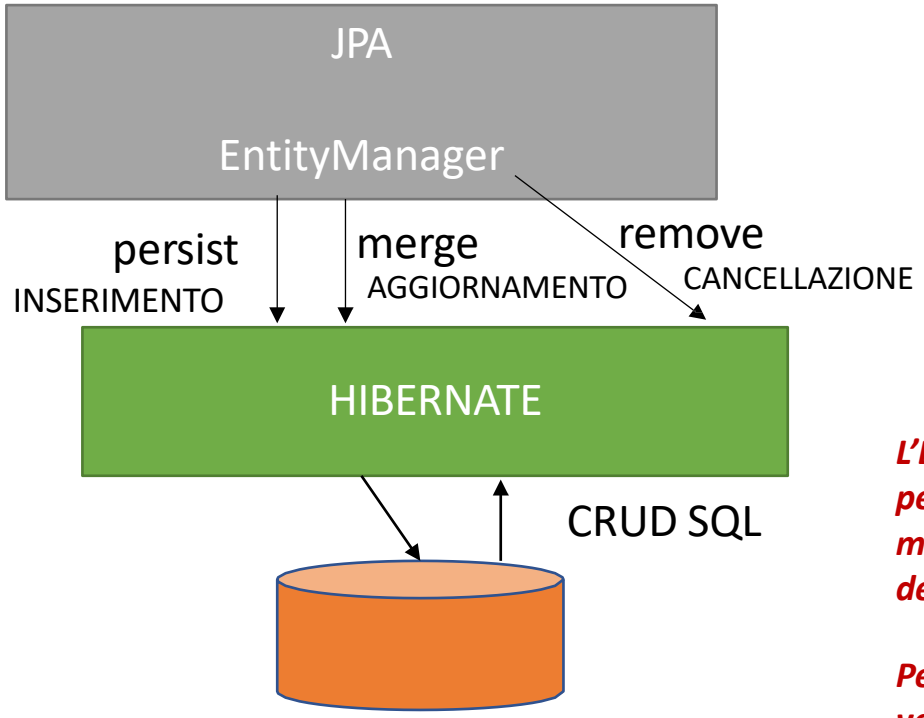
BOTTOM UP (ESISTE GIA' LA STRUTTURA FISICA DEL DATABASE E PER OGNI TABELLA MAPPIAMO CON LE ENTITY) - DB FIRST

TOP DOWN (CREIAMO LE ENTITY JPA IN MANIERA TALE CHE HIBERNATE COSTRUISCA LA STRUTTURA FISICA DEL DATABASE) - CODE FIRST

ESISTONO 2 PERSISTENCE PROVIDER IN GRADO DI TRADURRE CODICE JPA IN CODICE SQL:

- ECLIPSE LINK
- HIBERNATE

L' INTERMEDIARIO NELLA COMUNICAZIONE TRA JPA E HIBERNATE è l'EntityManager, Interfaccia JPA



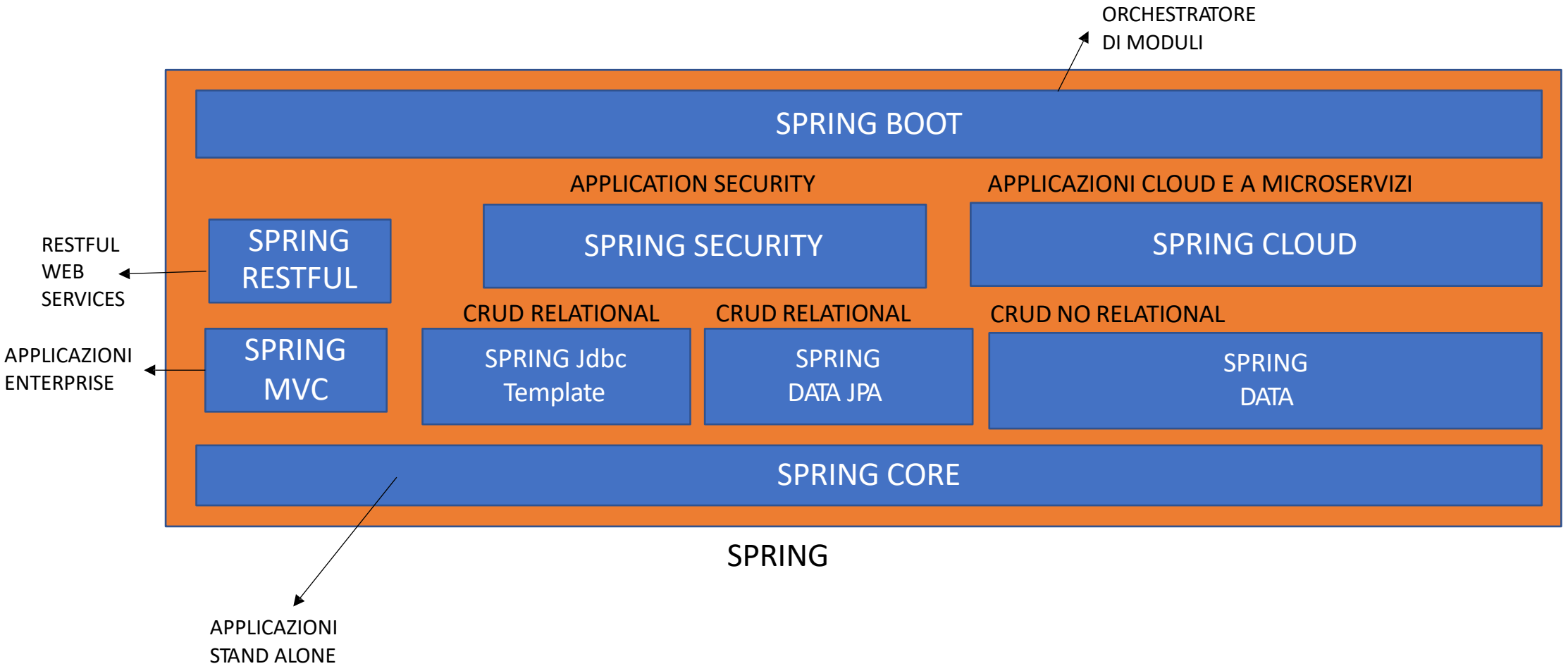
L'EntityManager è una Interfaccia JPA che espone una serie di metodi tra cui:
persist-→>> chiede ad Hibernate di effettuare una operazione di insert
merge→>> chiede ad Hibernate di effettuare una operazione di update
delete→>> chiede ad Hibernate di effettuare una operazione di cancellazione

Per le operazioni di lettura occorre utilizzare JPQL, pseudolinguaggio Java che verrà anch'esso tradotto da Hibernate in codice sql

FRAMEWORK = INSIEME DI API CHE NECESSITANO DI UN RUNTIME ENVIRONMENT FORNITO DAL FRAMEWORK STESSO PER ESSERE ESEGUITE

SPRING = FRAMEWORK JAVA **MODULARE** CHE CONSENTE L'IMPLEMENTAZIONE DI APPLICAZIONI STAND ALONE E APPLICAZIONI ENTERPRISE.
MODULARE SIGNIFICA CHE SPRING FORNISCE UN CERTO NUMERO DI MODULI APPLICATIVI OGNUNO DEI QUALI E' UN INSIEME DI API ED E' DEDICATO AD UNO SPECIFICO SCOPO APPLICATIVO:

SPRING BOOT E' UN MODULO «SINGOLARE» IN QUANTO NON E' PREPOSTO AD UNO SCOPO APPLICATIVO, MA PIUTTOSTO SERVE COME ORCHESTRATORE DI MODULI, FAVERONDONE LA COMUNICAZIONE TRAMITE LA SCRITTURA DI CODICE SNELLO E STRINGATO.



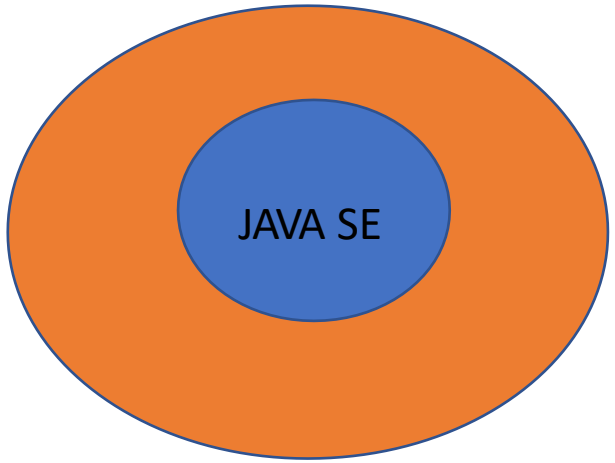
SPRING

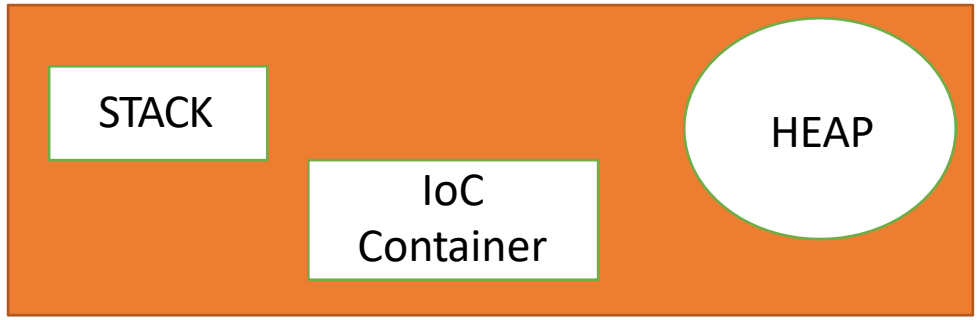
PLUG-IN

COMPILER + SPRING JVM

COMPILER + JVM

Spring Core



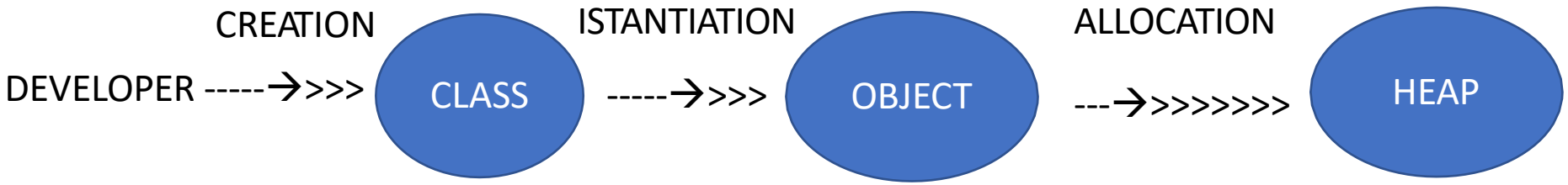


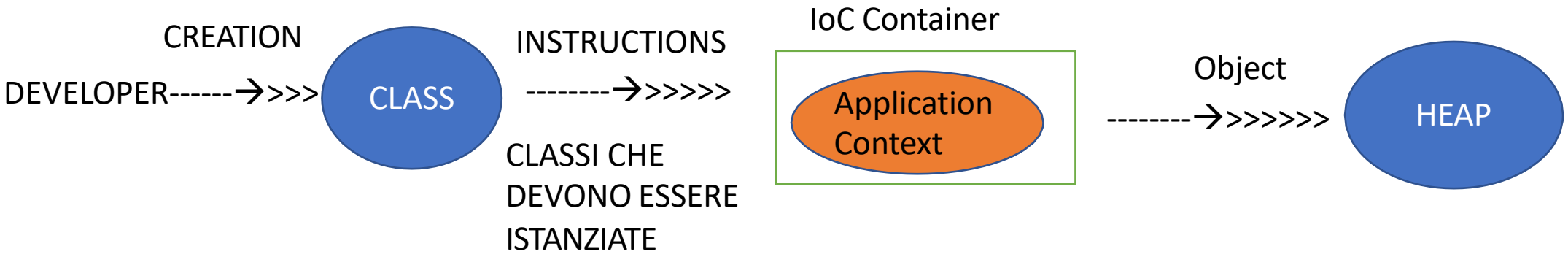
SPRING JVM

LA SPRING JVM E' IL RUNTIME ENVIRONMENT NECESSARIO ALL'ESECUZIONE DELLE APPLICAZIONI SPRING

SPRING JVM = JVM COMPLESSA (AUGMENTED) = STACK + HEAP + IoC Container

PROCESSO DI SVILUPPO JAVA SE





L'ApplicationContext istanzia e alloca all'interno dell'HEAP tutti gli Oggetti che derivano da Classi che sono annotate con le Stereotype Annotations (Spring Beans)

PROCESSO SVILUPPO SPRING CORE

SPRING BOOT

- In ogni Applicazione Spring Boot deve essere presente una classe annotata con l'annotation @SpringBootApplication(tale classe rappresenta l'ApplicationContext che verrà eseguita a Runtime come Oggetto all'interno dell'IoC Container);
- In ogni Applicazione Spring Boot deve essere presente un file di nome application.properties all'interno del quale si possono dare istruzioni al modulo;
- Ogni Applicazione ha nativamente dentro di sé un Embedded Web Container (Tomcat)
- In ogni applicazione Spring Boot è possibile selezionare, perché gia' pronti (built-in), gli STARTERS

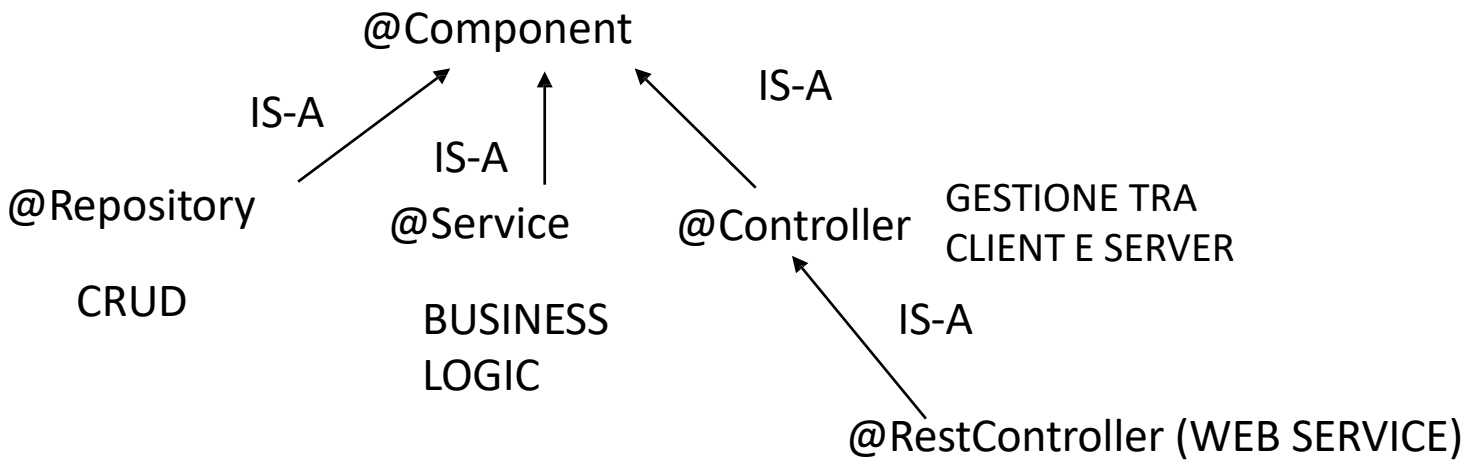
UNO STARTER SPRING BOOT è UN PACCHETTO DI DIPENDENZE MAVEN

ESEMPI DI STARTERS:

SPRING WEB = SPRING CORE + SPRING MVC + SPRING RESTFUL

SPRING DATA JPA = JPA + SPRING DATA JPA + HIBERNATE

Stereotype Annotations (FORNITE DA SPRING CORE) e relativa relazione di parentela



@Autowired è una annotation di DI Spring, tramite la quale è possibile «chiedere» all'ApplicationContext che viene eseguito nell'IoC Container di istanziare le Classi riconosciute come Spring Bean dal Framework, ovvero le Classi che sono annotate con una delle Stereotype Annotation. Le classi non annotate con le Stereotype Annotations sono istanziabili alla «vecchia maniera» ma non tramite DI.

CONCETTO DI TRANSAZIONE IN UN DATABASE RELAZIONALE

UNA TRANSAZIONE E' UNA O UN INSIEME DI OPERAZIONI CHE DEVONO ESSERE ESEGUITE TUTTE CORRETTAMENTE SUL DATABASE PENA IL ROLLBACK (RIPRISTINO) ALLO STATO PRECEDENTE (IN PRATICA O TUTTE LE OPERAZIONi VANNO BENE O TUTTE VANNO MALE)

AFFINCHE' HIBERNATE COMPIA CORRETTAMENTE LE OPERAZIONI SUL DATABASE OCCORRE COMUNICARGLI IN MANIERA ESPLICITA CHE E' STATA APERTA UNA CONNESSIONE AL DATABASE

TALE COMUNICAZIONE AVVIENE TRAMITE L'USO DELL'ANNOTATION @Transactional

ESISTONO DUE TIPI DI ANNOTATION @Transactional

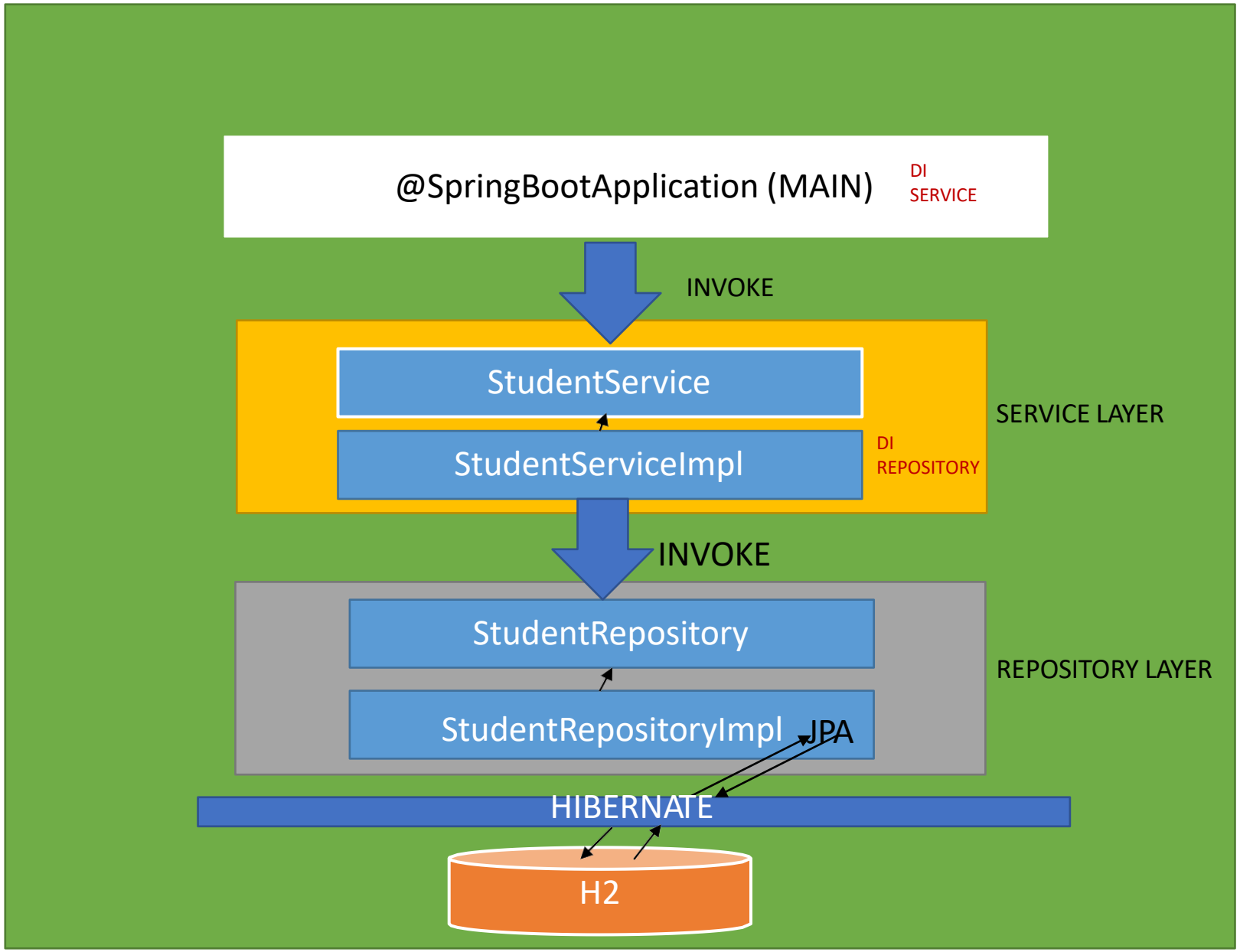
@Transactional (package javax.transaction) --→>>> FORNITA DA JAVA EE

@Transactional (package org.springframework…)-→>> FORNITA DA SPRING

LA PRIMA SUPPORTA SOLO TRANSAZIONI LOCALI, OVVERO TRANSAZIONI COMPOSTE DA OPERAZIONI CHE VENGONO ESEGUITE SU UN SOLO DATABASE

LA SECONDA (BEST PRACTICE) SI COMPORTA COME UNA TRANSAZIONE LOCALE O GLOBALE (E' QUINDI PIU' FLESSIBILE COME SOLUZIONE)

ARCHITETTURA APPLICATION SB_JPA_CRUD



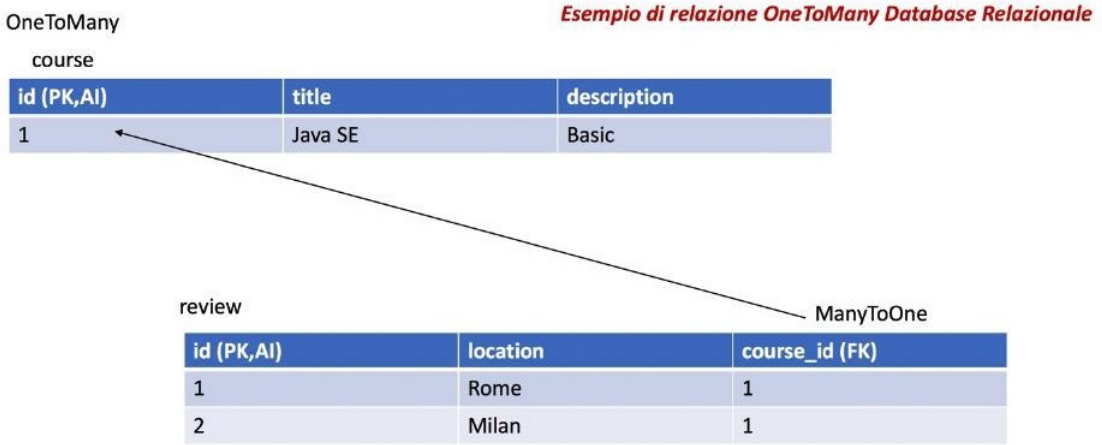
METOLOGIA DI MAPPING DI UNA RELAZIONE OneToMany tramite JPA

1.Implementazione delle due Entity che sono legate da relazione OneToMany e dichiarazione delle variabili di istanza che rappresentano le colonne «proprie» di ogni singola tabella corrispondente; dichiarazione dei metodi set e get e costruttori.

2. Rappresentazione del verso OneToMany
Nella JPA Entity (Entity padre) che rappresenta il verso OneToMany occorre dichiarare una variabile di istanza del tipo Collection Interface tipizzata con il tipo dell'Entity figlia (ovvero quella che rappresenta il vero ManyToOne); tale variabile deve essere preceduta dall'annotation @OneToMany; l'annotation @OneToMany deve contenere un parametro di nome mappedBy valorizzato con l'esatto nome della variabile di istanza usata sotto l'annotation @ManyToOne della Entity figlia; dichiarare i corrispondenti metodi set e get.

3. Rappresentazione del verso ManyToOne
Nella JPA Entity (Entity figlia) che rappresenta il verso ManyToOne occorre dichiarare una variabile di istanza del tipo dell'Entity padre, dichiarare i relativi metodi di set e get.
Tale variabile corrisponde alla fk della tabella figlia (Hibernate crea direttamente un campo come fk che ha il nome della variabile usata nell'Entity filia + _nome della pk dell'Entity padre.

RELAZIONE OneToMany Database Relazionale



Una relazione OneToMany si applica creando un campo sulla tabella che rappresenta il verso ManyToOne al quale si assegna la caratteristica di Foreign Key (tale campo deve referenziare la Primary Key della tabella che rappresenta il Verso OneToMany)

La tabella che contiene la FK è la tabella CHILD (PARENT).
La tabella che non contiene la PK è la tabella FATHER.

MODALITA' DI DATA FETCHING DA PARTE DEL PERSISTENCE PROVIDER PER TABELLE PADRE – FIGLIO (RELAZIONE OneToMany)

NEL CASO IN CUI VENGA RICHIESTO AD UN PERSISTENCE PROVIDER DI RECUPERARE DAL DATABASE INFORMAZIONI PRESENTI SU UNA TABELLA PADRE DI UN'ALTRA (TABELLA CHE RAPPRESENTA IL VERSO OneToMany) IL RECUPERO DELLE INFORMAZIONI PUO' ESSERE EFFETTUATO IN DUE DIVERSE MODALITA':

- **LAZY**

IL PERSISTENCE PROVIDER RECUPERA SOLO LE INFORMAZIONI STRETTAMENTE RICHIESTE DELLA TABELLA PADRE E LE CARICA NELLA SUA MEMORIA (CACHE) E NON RECUPERA ANCHE EVENTUALI INFORMAZIONI RELATIVE A TABELLE FIGLIE ASSOCIATE ALLA TABELLA PADRE. TALI INFORMAZIONI VERRANNO RECUPERATE DAL PERSISTENCE PROVIDER SOLO SE STRETTAMENTE RICHIESTO.
L'apertura di una connessione, l'esecuzione di una qualsiasi operazione di crud, e la chiusura della connessione, rappresentano per il Persistence Provider una SESSIONE TRANSAZIONALE.
Se chiediamo successivamente al PP di recuperare delle informazioni di una tabella figlio relative ad una info della tabella Padre, il Persistence Provider apre una nuova sessione transazionale, cioè apre una connessione, tenta di fare questa operazione, e poi dovrebbe chiudere la connessione.
Ogni Persistence Provider di default non è STATEFUL, ovvero non mantiene memoria di operazioni effettuate in diverse SESSIONI TRANSAZIONALI -→>> nel momento in cui chiediamo di recuperare info relative ad una sessione transazionale precedente Hibernate restituisce una LazyException.

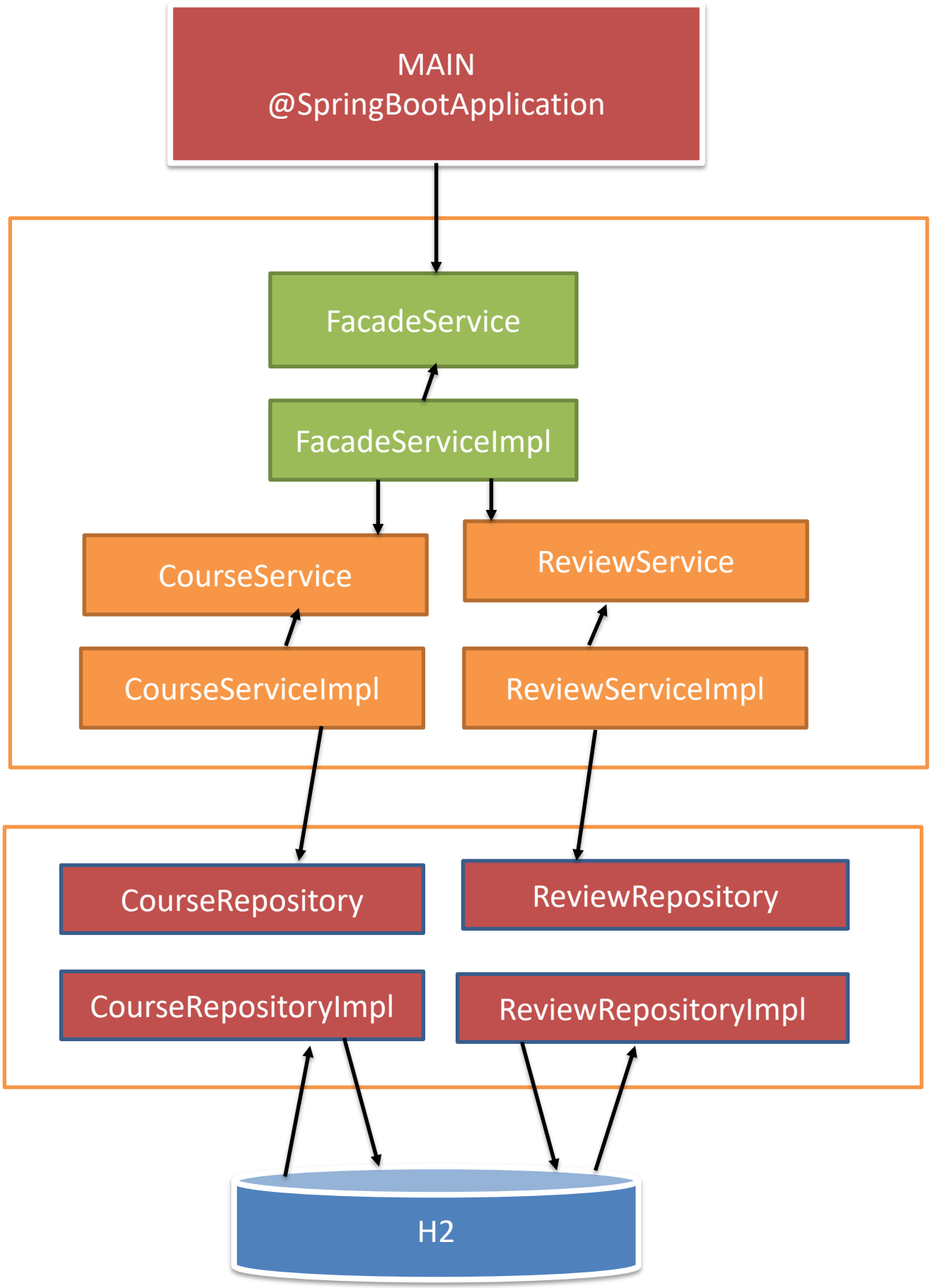
- **EAGER**

IL PERSISTENCE PROVIDER RECUPERA LE INFORMAZIONI RELATIVE ALLA TABELLA PADRE (QUELLE CHE GLI CHIEDIAMO) MA VERIFICA ANCHE SE ESISTONO NELLA TABELLA FIGLIA INFORMAZIONI ASSOCIATE ALLE INFO DELLA TABELLA PADRE E, NEL CASO IN CUI CI SIANO, RECUPERA ANCHE QUELLE E LE CONSERVA NELLA SUA MEMORIA (CACHE).
SE VIENE RICHIESTO DI RECUPERARE IN UNA SESSIONE TRANSAZIONALE DIFFERENTE QUALCHE INFO DELLA TABELLA FIGLIO IN QUESTO CASO IL PERSISTENCE PROVIDER CI RIUSCIRA' IN QUANTO HA CARICATO IN CACHE NELLA PRECEDENTE SESSIONE TRANSAZIONALE LE INFORMAZIONI DELLA TABELLA FIGLIA (PUR SE NON ESPLICITAMENTE RICHIESTO)

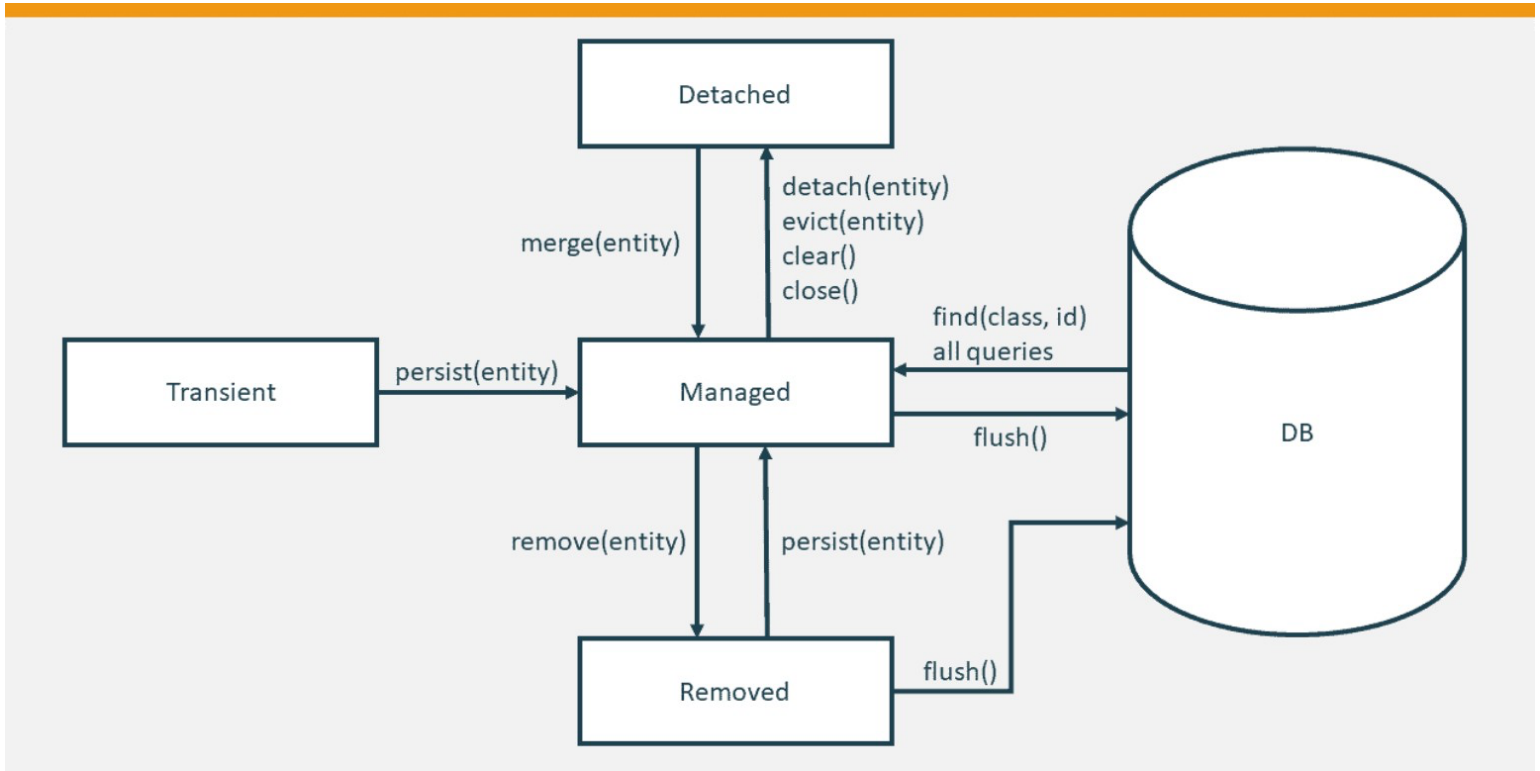
REGOLA JPA (DI CUI IL PERSISTENCE PROVIDER E' A CONOSCENZA):

Tutte le annotation JPA che terminano in …….**ToMany** hanno come modalità di fetching (info per il PP) impostata di default la modalità LAZY
Tutte le annotation JPA che terminano in ……**ToOne** hanno come modalità di fetching (info per il PP) impostata di default la modalità **EAGER**

ARCHITETTURA APPLICAZIONE SB_JPA_CRUD_ONE_TO_MANY



Ogni OGGETTO Entity JPA ha un ciclo di vita caratterizzato dalle seguenti fasi:



Un Oggetto Entity si trova in stato New quando l'Entity viene istanziata dal Programmatore; l'Oggetto in stato NEW viene allocato nell'HEAP; Hibernate non legge dall'HEAP, pertanto è ignaro dell'esistenza dell'Oggetto Entity in stato NEW (TRANSIENT).
L'invocazione del metodo persist fa si che un Oggetto Entity passi dallo stato NEW allo stato MANAGED(ATTACHED). Quando l'Oggetto Entity si trova in stato MANAGED viene fatto un Clone dell'Oggetto che si trovava nell'HEAP, tale clone viene collocato in un'area di memoria chiamata PeristenceContext nel quale Hibernate vede la sua esistenza e compie le operazioni richieste (inserimento).

RELAZIONE MANY TO MANY LATO DATABASE RELAZIONALE

Esempio di relazione ManyToMany su un database relazionale

course

id(int) PK AI	title (varchar)	description (varchar)
1	Java	Basic
2	Angular	Front End

course_student

course_id	student_id
1	1
1	2
2	1
2	2

JOIN TABLE

course_id = fk verso la tabella course
student_id= fk verso la tabella student
course_id + student_id = pk

student

id(int) PK AI	firstname (varchar)	lastname (varchar)	age (int)
1	Marco	Rossi	25
2	Gianluca	Verdi	26

RELAZIONE MANY TO MANY LATO JPA

Una Relazione JPA Many To Many si può implementare in due differenti modalità:

1. Creare 3 Entity, due che mappano le tabelle legate da relazione, una che mappa la JOIN TABLE
2. Creare 2 Entity, che mappano entrambe le tabelle legate da relazione. In una Entity occorre definire una variabile di istanza del tipo struttura dati dell'altra Entity. Tale variabile va preceduta dall'annotation @ManyToMany e parametrizzata con mappedBy valorizzato al nome di variabile di istanza dell'altra Entity (anch'essa preceduta da annotation @ManyToMany). Nell'Entity che non contiene il mappedBy occorre inserire anche il riferimento alla JoinTable tramite apposizione di una specifica annotation (@JoinTable)

JPA = JAVA EE API

SPRING DATA JPA = MODULO DI SPRING CHE RAPPRESENTA UN'EVOLUZIONE ULTERIORE DI JPA

CRUD

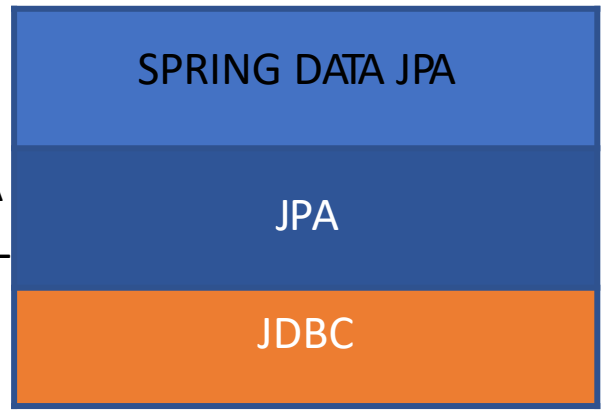
PURE JAVA

EntityManager - persist, merge, remove

JAVA

lettura

JPQL



PLUGIN JPA

RIDUZIONE MOLTO FORTE DELL' IMPEDENCE MISMATCHING RISPETTO ALLA STRUTTURA FISICA

PLUG-IN JDBC

richiede intermediari tra applicazione e database

Persistence Provider

non richiede intermediari tra applicazione e database

IMPEDENCE MISMATCHING RISPETTO ALLA STRUTTURA FISICA

SQL

PROCESSO DI SVILUPPO SPRING BOOT - JPA :

1. CONFIGURAZIONE DEGLI STARTERS DI PROGETTO (pom.xml)
2. CONFIGURAZIONE DELL' application.properties
3. CREAZIONE DELLE ENTITY JPA
4. CREAZIONE DELLO STRATO DI REPOSITORY CON CLASSE E INTERFACCIA
5. CREAZIONE DELLO STRATO DI SERVICE PER LA LOGICA DI BUSINESS CON CLASSE E INTERFACCIA
6. INVOCAZIONE DELLO STRATO DI SERVICE NEL MAIN

PROCESSO DI SVILUPPO SPRING BOOT – SPRING DATA JPA :

1. CONFIGURAZIONE DEGLI STARTERS DI PROGETTO (pom.xml)
2. CONFIGURAZIONE DELL' application.properties
3. CREAZIONE DELLE ENTITY JPA
4. CREAZIONE DELLO STRATO DI REPOSITORY CON INTERFACCE
5. CREAZIONE DELLO STRATO DI SERVICE PER LA LOGICA DI BUSINESS CON CLASSE E INTERFACCIA
6. INVOCAZIONE DELLO STRATO DI SERVICE NEL MAIN

SPRING DATA JPA FORNISCE UNA API DI TIPO INTERFACCIA JpaRepository che contiene la firma di alcuni metodi che rappresentano delle operazioni di CRUD già pronte all'uso :

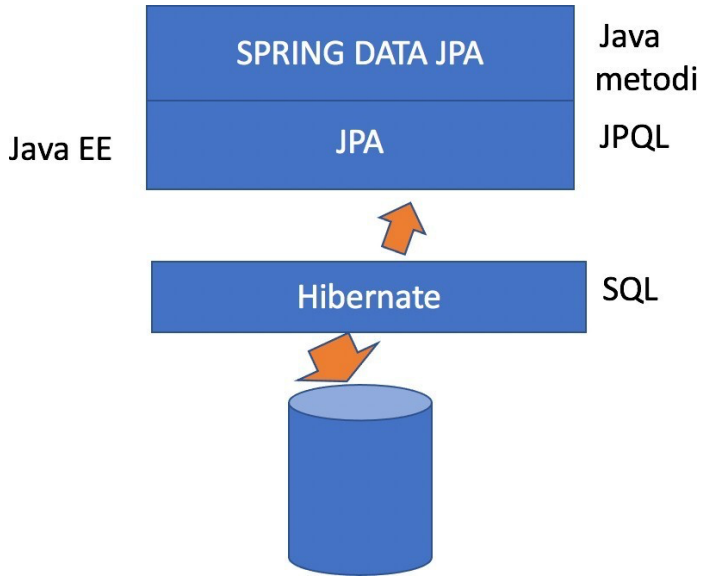
save (operazioni di inserimento e di aggiornamento)
deleteById (operazione di cancellazione per chiave primaria)
findAll (consente di eseguire operazioni di lettura integrale sul database)

PROCEDURA CLASSICA : CREARE UNA INTERFACCIA CUSTOM CHE ESTENDE JpaRepository

L'interfaccia JpaRepository secondo le specifiche di Spring Data JPA deve essere tipizzata con due tipi:
1. tipo di una Entity
2.tipo della chiave primaria (direttamente il tipo se la chiave primaria è di tipo Object, Wrapper del tipo se la chiave primaria è di tipo primitivo)

Tipizzare JpaRepository con una Entity vuol dire
rendere abile JpaRepository ad eseguire operazioni
Di CRUD sull'Entity di pertinenza

Spring Data Jpa rappresenta di fatto un superset (plug in) della specifica JPArilasciata dalla Java EE

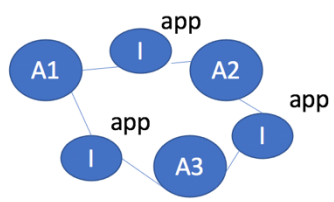


TIPOLOGIE DI ARCHITETTURE IT

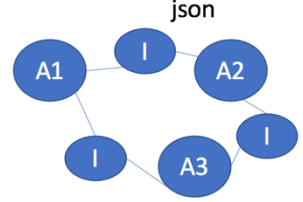
1. ARCHITETTURA IT TRADIZIONALE = LOGICA A SPAGHETTI
2. ARCHITETTURA SOA (SERVICE ORIENTED ARCHITECTURE)
   Sottocategoria Architettura a Microservizi

In una architettura tradizionale applicazioni implementate in diversi linguaggi si «parlano» attraverso interfacce applicative.
In una architettura SOA applicazioni implementate in diversi linguaggi si «parlano» attraverso interfacce non applicative
(standard «machine readable» come xml e json)

ARCHITETTURA IT TRADIZIONALE



xml--------→>>>>wsdl
json

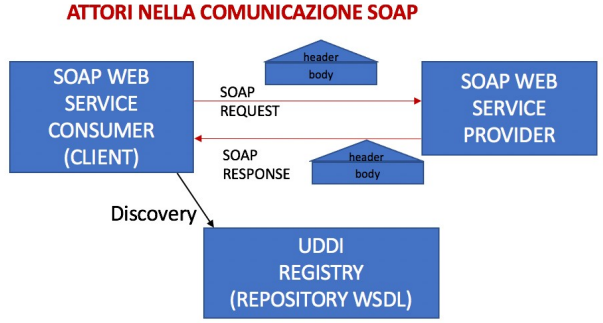


ARCHITETTURA IT SOA

SERVIZIO = APPLICAZIONE IN GRADO DI ESPORSI AL MONDO ESTERNO SOTTO FORMA DI INTERFACCIA NON APPLICATIVA (IN MODALITA' AGNOSTICA RISPETTO AI DETTAGLI IMPLEMENTATIVI) OVVERO ATTRAVERSO UNO STANDARD UNIVERSALMENTE NOTO A TUTTE LE APPLICAZIONI E A TUTTE LE MACCHINE

SOAP WEB SERVICE PROVIDER = WEB SERVICE PROVIDER CHE SI ESPONE SOTTO FORMA DI WSDL
RESTFUL WEB SERVICE PROVIDER = WEB SERVICE PROVIDER CHE SI ESPONE SOTTO FORMA DI JSON

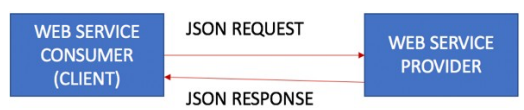
ATTORI NELLA COMUNICAZIONE SOAP



UDDI REGISTRY = UNIVERSAL DESCRIPTOR DISCOVERY INTERFACE = REPOSITORY WSDL

SOAP = Simple Object Access Protocol

ATTORI NELLA COMUNICAZIONE RESTFUL



OGNI SERVIZIO RESTFUL SI ESPONE SOTTO FORMA DI RESOURCE

resource = URI (operation name) + http verb

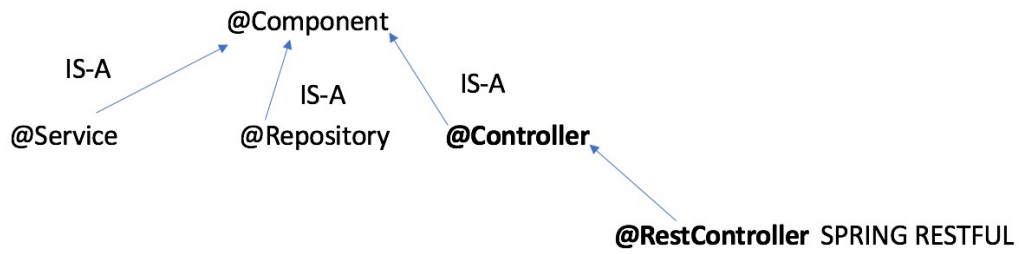
TECNOLOGIE JAVA PER L'IMPLEMENTAZIONE DI WEB SERVICE PROVIDER

JAX-WS = Java API XML for Web Service Provider ------→>>>>>JAVA EE API ----→>>>SOAP WEB SERVICE PROVIDER

JAX-RS = Java API XML for Restful Web Service Provider------→>>>>>JAVA EE API ----→>>>RESTFUL WEB SERVICE PROVIDER

SPRING RESTFUL = Modulo Spring ----------→>>>> RESTFUL WEB SERVICE PROVIDER

GERARCHIA EREDITARIETA' STEREOTYPE ANNOTATIONS



@Service is a Component da delegare alla BUSINESS LOGIC

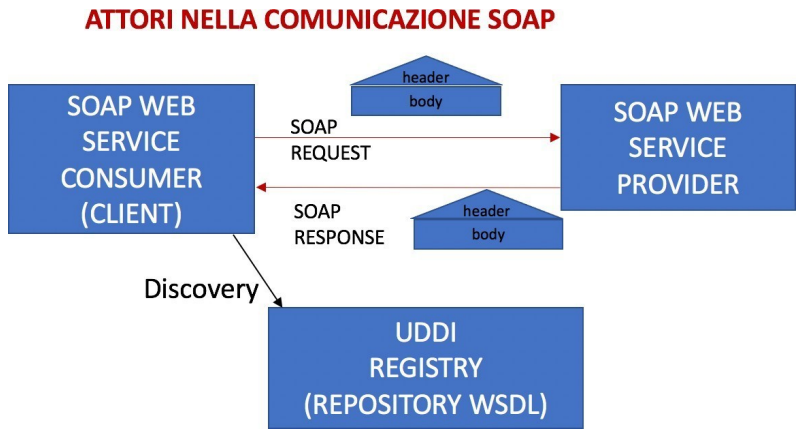
@Repository is a Component da delegare all'implementazione della logica di CRUD

@Controller is a Component da delegare alla gestione di richieste e risposte

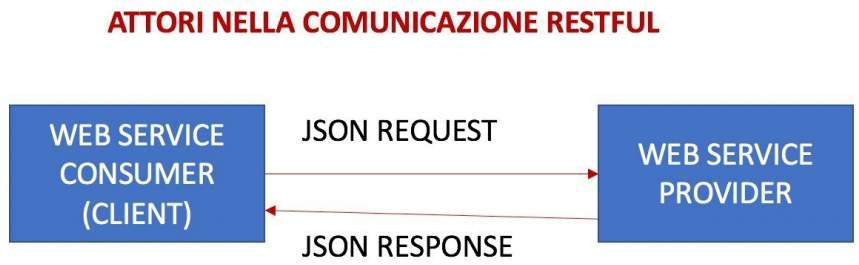
@RestController is a Component introdotta da Spring Restful per esporre una classe sotto forma di Restful Web Service Provider, quindi per esporre resources

Esistono tre diverse tipologie di ARCHITETTURA SOA:

1. ARCHITETTURA SOAP



2. ARCHITETTURA REST



OGNI SERVIZIO RESTFUL SI ESPONE SOTTO FORMA DI RESOURCE

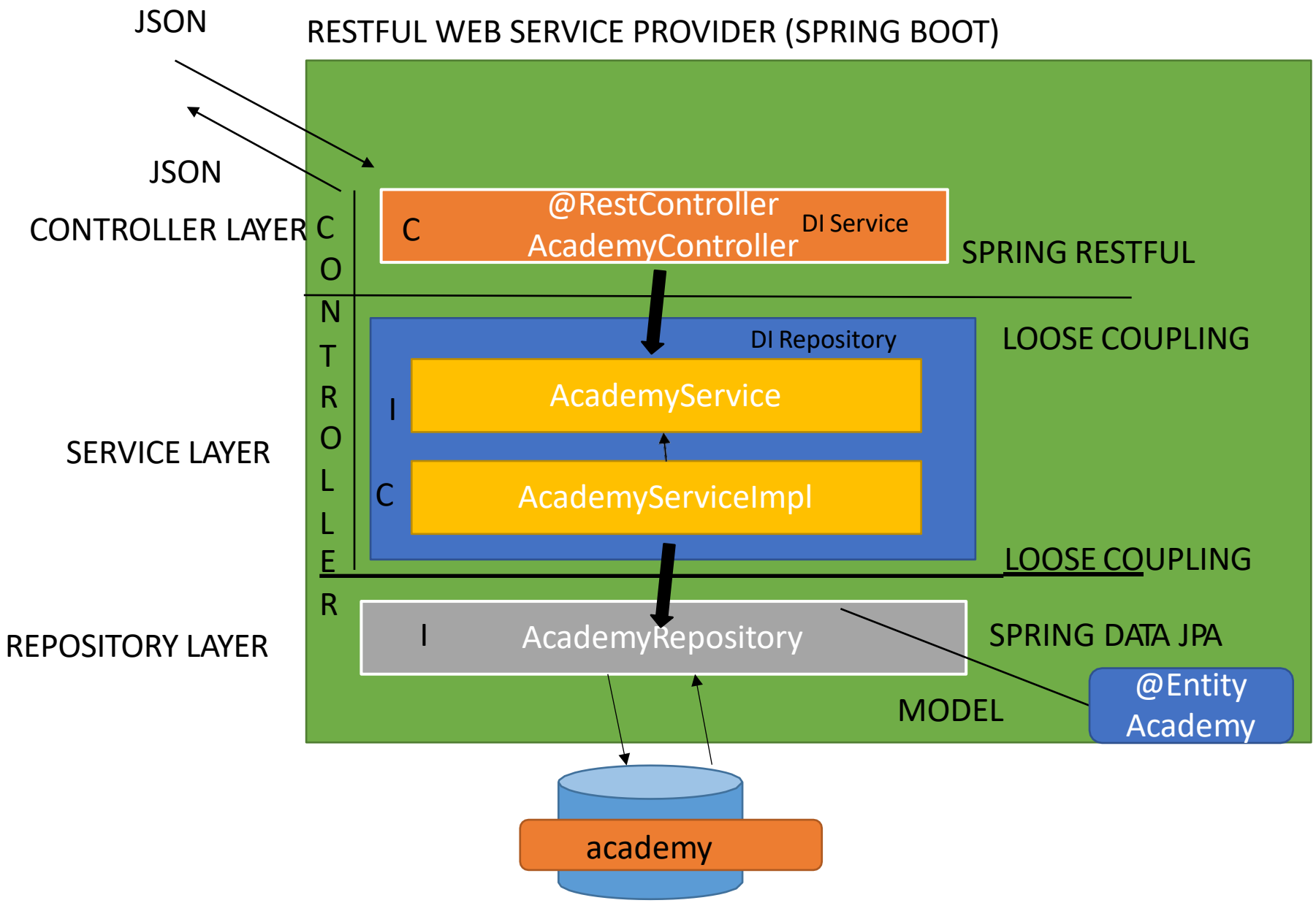
resource = URI (operation name) + http verb

BEST PRACTICE IN UNAARCHITETTURA REST: usare il verbo http put per le operazioni di update, in quanto PUT è idempotente (POST non lo è)

OPERAZIONE IDEMPOTENTE = OPERAZIONE IRRIPETIBILE

}

ARCHITETTURA APPLICAZIONE Academy_Restful_Web_Service_Provider



OPERAZIONI ESPOSTE DALL'APPLICAZIONE SB_ACADEMY_RESTFUL_WEB_SERVICE_PROVIDER

GET /rest/api/academies JSON --→>> tutte le academies

GET /rest/api/academies/code JSON --→>> academy con un determinato code

POST /rest/api/academies JSON---→>> nuova academy inserita sul database

PUT /rest/api/academies JSON---→>> aggiornamento di una academy esistente sul database

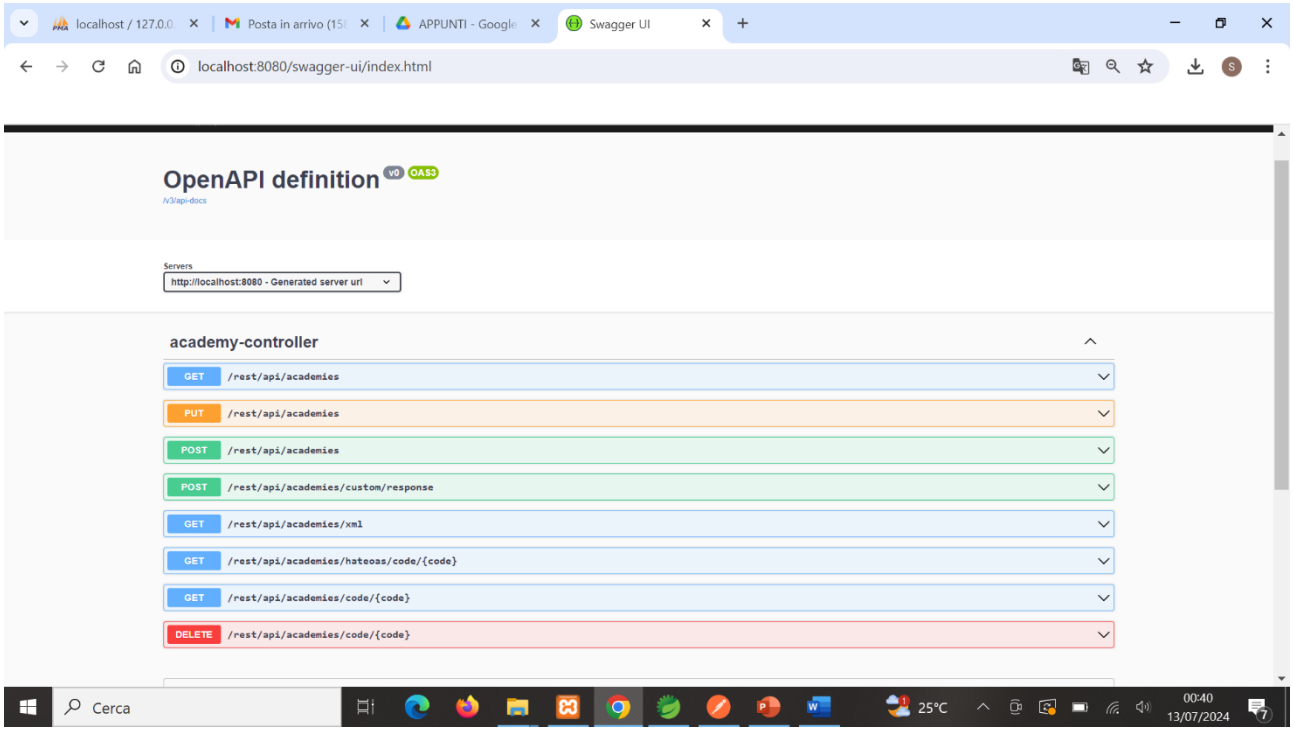
DELETE /rest/api/academies/code JSON→>> cancellazione di una academy esistente sul database

DOCUMENTAZIONE DI UN RESTFUL WEB SERVICE PROVIDER (@RestController)

E’ POSSIBILE DOCUMENTARE UN RESTFUL WEB SERVICE PROVIDER CONFIGURANDO ALL’INTERNO DELL’APPLICAZIONE LA SEGUENTE DIPENDENZA MAVEN:

```
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
   <version>2.0.2</version>
</dependency>
```

LA DOCUMENTAZIONE SARA’ SUCCESSIVAMENTE VISUALIZZABILE SUL SEGUENTE LINK:
http://localhost:8080/swagger-ui/index.html



VALIDAZIONE DI UN JSON

E' POSSIBILE IMPLEMENTARE MECCANISMI DI VALIDAZIONE UTILIZZANDO IL SEGUENTE STARTER SPRING BOOT, IMPOSTANDO REGOLE DI VALIDAZIONE ED UTILIZZANZO L'ANNOTATION @Valid IN RICEZIONE DI UN JSON PROVENIENTE DAL CONSUMER

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

GESTIONE CENTRALIZZATA DELLE ECCEZIONI

SI PUO' CENTRALIZZARE LA GESTIONE DELLE ECCEZIONI ALL'INTERNO DI UN APPLICAZIONE WEB SERVICE PROVIDER CREANDO UNA CLASSE ANNOTATA CON @RestController. TALE CLASSE RAPPRESENTA UN SUPERVISOR, UN LISTENER IN GRADO DI PORSI IN ASCOLTO SU TUTTE LE ECCEZIONI CHE SI POSSONO VERIFICARE ALL'INTERNO DEI @RestController DI PROGETTO. TRAMITE L'ANNOTATION @ExceptionHandler E' POSSIBILE ISTRUZIONARE IL SUPERADVISOR IN MANIERA TALE CHE, AL VERIFICARSI UNA DETERMINATA ECCEZIONE, VENGA RESTITUITO UN JSON CON UN MESSAGGIO CUSTOM.

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.context.request.WebRequest;

import com.sistemi.imformativi.exception.ErrorMessage;

@RestControllerAdvice
public class ExceptionAdvice {

   @ExceptionHandler(MethodArgumentNotValidException.class)
   @ResponseStatus(value= HttpStatus.BAD_REQUEST)
   public ErrorMessage notValidExceptionHandler(Exception ex, WebRequest request) {

      ErrorMessage message = new ErrorMessage(HttpStatus.BAD_REQUEST.value(), new Date(),
            "code must of be of 4 characters", request.getDescription(false));

      return message;

   }

}
```

PATTERN HATEOAS = HYPERMEDIA AS THE ENGINE OF THE APPLICATION STATE = E' PREFERIBILE RESTITUIRE AL CONSUMER RISPOSTE CHE NON CONTENGANO SOLO IL CONTENUTO INFORMATIVO, MA UNA RAPPRESNTAZIONE DELL'INTERO STATO DEL SERVIZIO REST, OVVErO L'INSIEME DELLE SUE OPERAZIONI REST. IN PRATICA IL PATTERN HATEOAS SI PONE L'OBIETTIVO DI RENDERE NAVIGABILI LE RISPOSTE, CHE CONTERRANNO DEGLI HYPERLINK, CIASCUNO DEI QUALI RIMANDI AS UNA SPECIFICA OPERAZIONE ESPOSTA.

PER IMPLEMENTARE IL PATTERN HATEOAS SPRING BOOT METTE A DISPOSIZIONE IL SEGUENTE STARTER

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

ESEMPIO DI IMPLEMENTAZIONE DEL PATTERN HATEOAS

```
import org.springframework.hateoas.server.mvc.WebMvcLinkBuilder;
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.linkTo;
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.methodOn;

@GetMapping("/hateoas/code/{code}")
        public EntityModel<Academy> getAcademyByCodeHateoas
(@PathVariable String code)  {

                Academy academy = academyService.getAcademyByCode(code);


                EntityModel<Academy> resource = EntityModel.of(academy);

                // http://localhost:8080/rest/api/academies
                WebMvcLinkBuilder linkTo = linkTo(methodOn(this.getClass()).getAcademies());

                // associazione di una etichetta al link
                resource.add(linkTo.withRel("all-academies"));

                return resource;

        }
```

Il RESTCONTROLLER SPRING RESTITUISCE DI DEFAULT JSON, MA PUO' ESSERE ABILITATO A RESTITUIRE ANCHE XML CONFIGURANDO ALL'INTERNO DEL PROGETTO LA SEGUENTE DIPENDENZA MAVEN:

```
<dependency>
<groupId>com.fasterxml.jackson.dataformat</groupId>
<artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

```
/*
* esempio di un metodo che restituisce al consumer un xml
*/
@GetMapping(path = "/xml", produces = MediaType.APPLICATION_XML_VALUE)
public List<Academy> getAcademiesXML()  {

        return academyService.getAcademies();

}
```

E' POSSIBILE CUSTOMIZZARE IL CONTENUTO DEL BODY DI RISPOSTA JSON ASSOCIANDOLO AD UNO SPECIFICO STATUS CODE DIVERSO DA QUELLO DI DEFAULT, TRAMITE L'API SPRING RESTFUL ResponseEntity

```
@PostMapping("/responseEntity")
public ResponseEntity<?> customSaveAcademy(@RequestBody Academy academy) {

        academyService.saveOrUpdateAcademy(academy);
        Map<String, String> responseMap = new HashMap<>();
        responseMap.put("save academy operation", "ok");

        return new ResponseEntity<>(responseMap, HttpStatus.CREATED);

}
```

E' POSSIBILE RESTRINGERE IL CONSUMO DELLE OPERAZIONI REST SOLO A CHI NE E' AUTORIZZATO, CONFIGURANDO ALL'INTERNO DEL PROGETTO LO STARTER SPRING SECURITY E CONFIGURANDO LE CREDENZIALI DI ACCESSO ALL'INTERNO DELL'APPLICATION.PROPERTIES (VEDI ESEMPIO SOTTOSTANTE)

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

```
spring.security.user.name=sa
spring.security.user.password=sa
```

PER OTTIMIZZARE LE PERFORMANCE DI HIBERNATE, E' POSSIBILE SETTARE LA CACHE DI SECONDO LIVELLO IN MANIERA TALE CHE COMPIA UNA OPERAZIONE DI QUERY UA SOLA VOLTA E SUCCESSIVAMENTE ATTINGA DALLA SUA CACHE SENZA RIESEGUIRLA SUL DATABASE. LA CACHE DI SECONDO LIVELLO SI PUO' IMPOSTARE TRAMITE UN CACHE PROVIDER (eh-cache, hazelcast, caffeine,ecc)

MEMO DEI PASSAGGI DA EFFETTUARE HIZELCAST:

1. CONFIGURAZIONE DELLE DIPENDENZE NEL pom.xml

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
<groupId>com.hazelcast</groupId>
<artifactId>hazelcast</artifactId>
</dependency>
<dependency>
<groupId>com.hazelcast</groupId>
<artifactId>hazelcast-spring</artifactId>
</dependency>
```

2. *Uso della annotation* @EnableCaching nella classe @SpringBootAplication e configurazione del Bean con assegnazione del nome alla cache

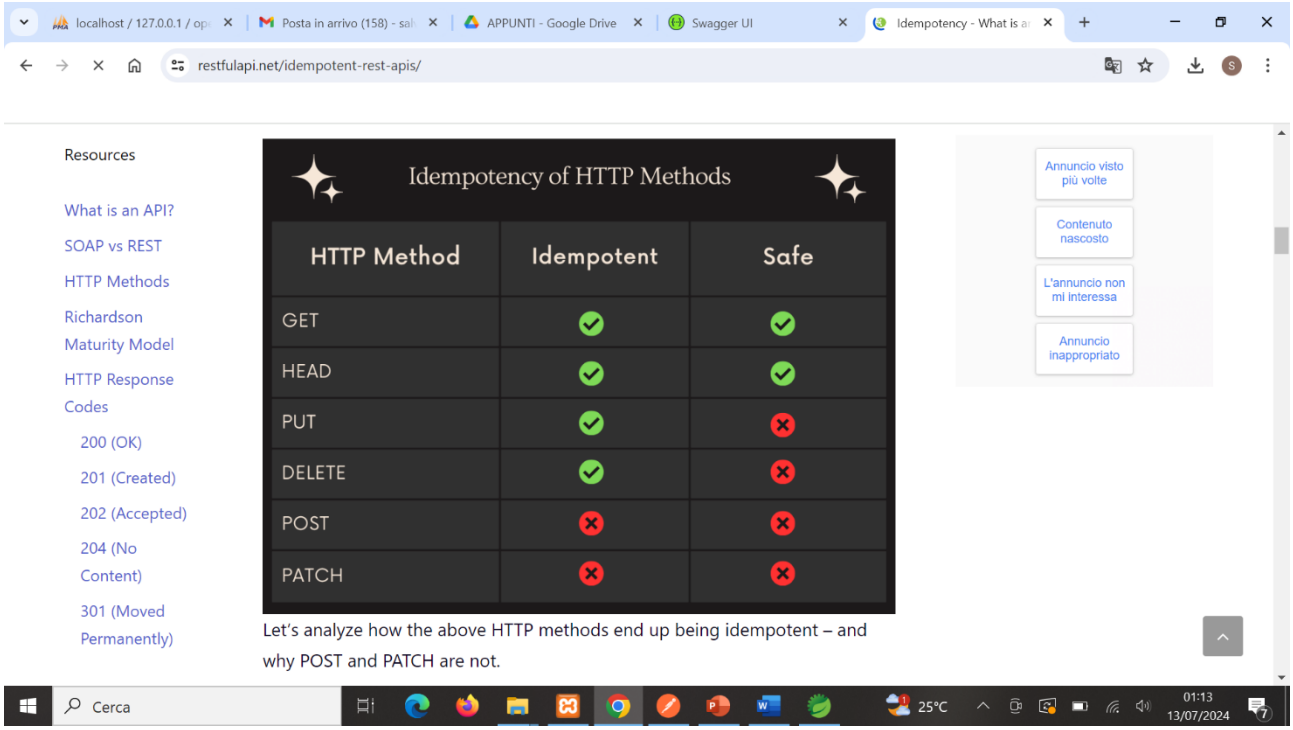
```
@Bean
	Config hazelCastConfig() {
		return new Config().setInstanceName("hazelcast-instance")
				.addMapConfig(new
MapConfig().setName("academies").setTimeToLiveSeconds(20));
}
```

3. Match del nome della cache tramite annotation @CacheConfig nella classe Service, e annotation @Cacheable sopra il metodo che si intende cachare.

```
package com.sistemi.informativi.service;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.cache.annotation.CacheConfig;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

import com.sistemi.informativi.entity.Academy;
import com.sistemi.informativi.repository.AcademyRepository;
import jakarta.persistence.EntityNotFoundException;
```

```
@Service
@CacheConfig(cacheNames="academies")
public class AcademyServiceImpl implements AcademyService {

	private AcademyRepository academyRepository;

	public AcademyServiceImpl(AcademyRepository academyRepository) {

		this.academyRepository = academyRepository;
	}



	@Cacheable(value = "code", key = "#code")
	@Override
	public Academy getAcademyByCode(String code) {

		return academyRepository.findById(code).orElseThrow(() -> new
EntityNotFoundException("not existing code"));

	}

}
```

CARATTERISTICHE DEI VERBI HTTP REST SPECIFICATI DALL'ARCHITETTURA REST



UNA OPERAZIONE REST E’ IDEMPOTENTE SE, PUR RIPETUTA PIU’ VOLTE IN MANIERA IDENTICA, NON PRODUCE SIDE EFFECTS SUL SISTEMA PRODICENDO LA CREAZIONE DI NUOVE RISORSE.

BEST PRACTICE: UTILIZZARE PUT (OPERAZIONE IDEMPOTENTE) PER AGGIORNAMENTO, POST (OPERAZIONE NON IDEMPOTENTE) PER L’INSERIMENTO.