

# Sviluppo del back-end di un'applicazione web con Spring Boot e STS

## Spring

Spring è un framework open-source, per lo sviluppo di applicazioni Web su piattaforma Java, ampiamente utilizzato in ambito industriale.

Offre diversi servizi distribuiti in vari moduli:

- Spring Core (modulo principale),
- modulo AOP (Aspect Oriented Programming),
- Data Access (persistenza su database),
- Inversion of Control (Dependency Injection),
- ModelAndViewController (per web app) e Remote Access

# Spring Boot

Spring Boot è un progetto Spring che ha lo scopo di rendere più semplice lo sviluppo e l'esecuzione di applicazioni Spring.

In genere, le applicazioni Spring Boot richiedono configurazione minima.

Spring Boot configura automaticamente Spring e le librerie di terze parti se possibile, permettendo agli sviluppatori di configurare solo il necessario.

Con a Spring Boot l'applicazione sarà distribuita usando un singolo file JAR (o WAR se richiesto) contenente tutto il necessario per essere eseguita (non è nemmeno necessario installare un server Tomcat a parte).

## Requisiti

- Java JDK 8
- MySQL Server
- Un IDE (raccomandato STS - Spring Tool Suite, un Eclipse modificato)
- Postman (per testing, per inviare messaggi POST al server), facoltativo
- Un approccio alternativo a quello qui esposto, basato su STS, si può trovare in:  
<https://spring.io/guides/gs/spring-boot/>

# Descrizione progetto

Utilizziamo dunque Spring Boot per creare il back-end di un portale web per la gestione dei risultati degli esami di un corso.

L'applicazione utilizza i seguenti moduli del framework *Spring*:

- Spring Data (JPA, Hibernate...) e JDBC, necessari per assicurare la persistenza delle risorse su un DBMS (MySQL)
- Spring MVC: per le chiamate REST che verranno utilizzate dal front-end
- Spring Security: per implementare un sistema RBAC, proteggere le risorse e gestire i permessi

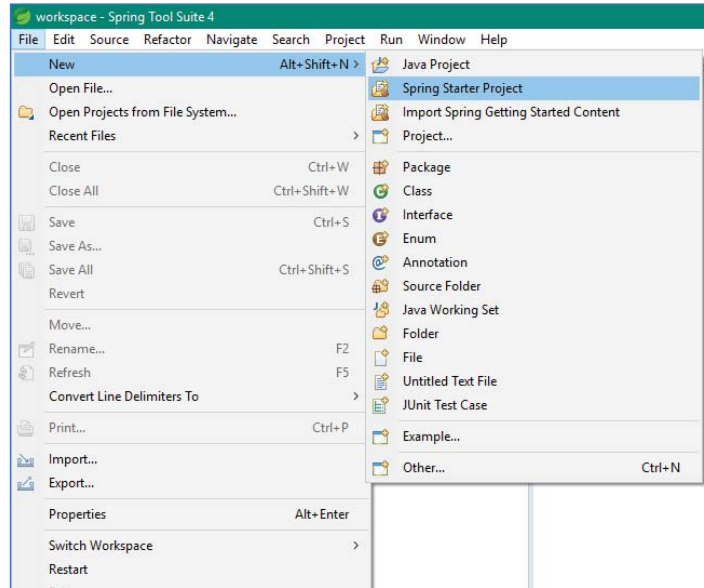
## Per iniziare

- Il portale *Spring Initializr* (<https://start.spring.io>) è uno strumento Web per creare progetti Spring Boot
- Inserendovi i dati del progetto e i moduli che esso usa, si può scaricare (uno *zip* con) un progetto Maven (o Gradle) pronto per essere importato in un IDE come STS
- Alternativa: operare solo dentro STS (che è in grado di interfacciarsi con *Initializr*)

The screenshot shows the Spring Initializr web application. At the top, it says "SPRING INITIALIZR bootstrap your application". Below this, there's a form to "Generate a" project. The "Maven Project" dropdown is selected, and "with" is followed by a "Java" dropdown. Below this, "and Spring Boot" is followed by a "2.1.1" dropdown. The "Project Metadata" section has "Artifact coordinates" with "Group" set to "exam-portal" and "Artifact" set to "portal". The "Dependencies" section has a search bar with "Web, Security, JPA, Actuator, Devtools..." and "Selected Dependencies" showing "Web", "JPA", and "MySQL". A "Generate Project" button is at the bottom right.

# Creare un progetto usando STS

- Spring Tool Suite è una installazione di Eclipse su misura per lo sviluppo Spring
- Integra i componenti necessari o utili per Spring
- Con STS si può creare un progetto Spring Boot direttamente dall'IDE
- Con la creazione guidata da un *wizard*, il progetto viene generato sul sito *Initializr*, scaricato e importato direttamente all'interno del workspace di STS



12/12/2020

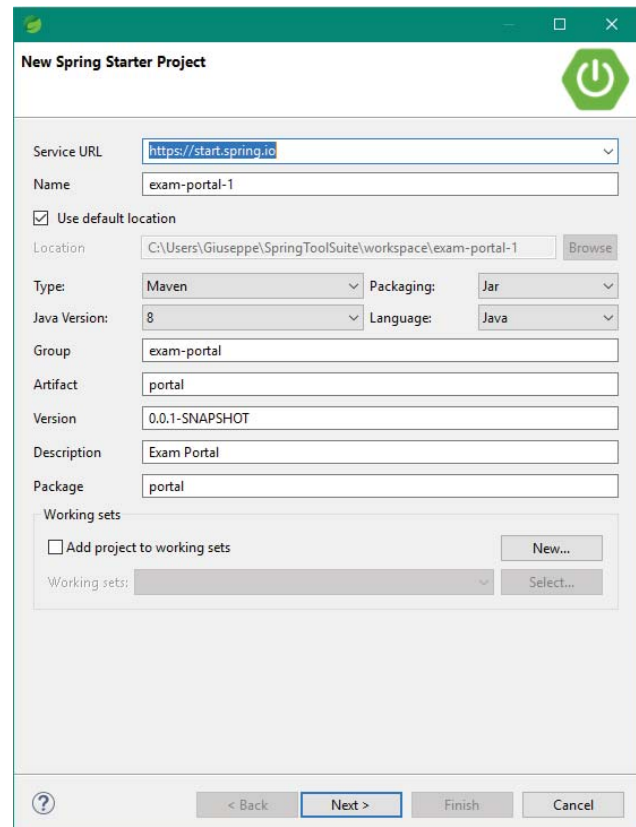
Web app SpringBoot con STS

6

## Creare un progetto usando STS (cont)

Vengono richiesti i dati del progetto da creare tra cui:

- nome
- una descrizione
- se usare *Maven* (come qui) o *Gradle* per la gestione delle dipendenze
- **ATTENZIONE DIRE QUI MA ANCHE DOPO DEL VINCOLO SUL NOME DEI PACKAGE!!!**



12/12/2020

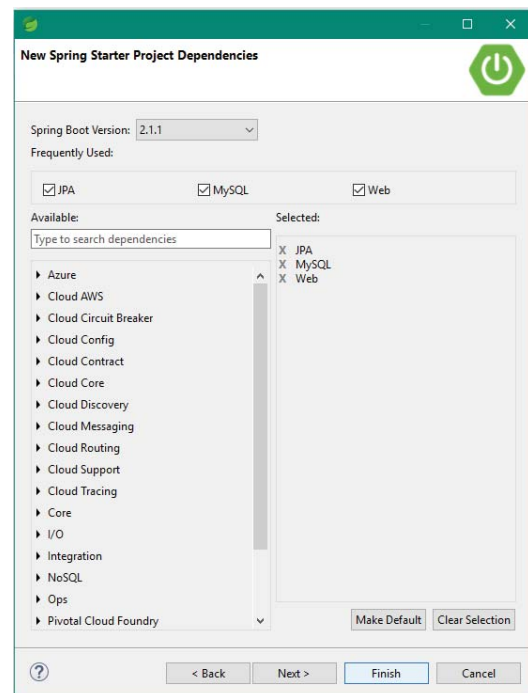
Web app SpringBoot con STS

7

# Creare un progetto usando STS (cont)

Occorre ancora indicare

- versione di Spring Boot da usare
- dipendenze del progetto, qui sono:
  - JPA (Java Persistence API)
  - MySQL come database
  - Web, per usare il modulo Spring MVC
  - NB: le dipendenze si possono inserire anche in seguito



12/12/2020

Web app SpringBoot con STS

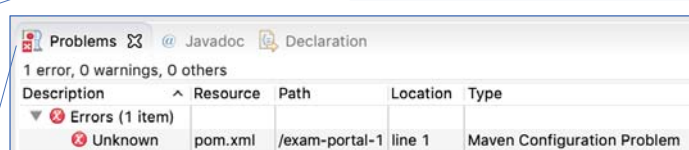
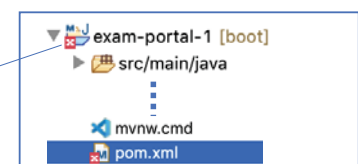
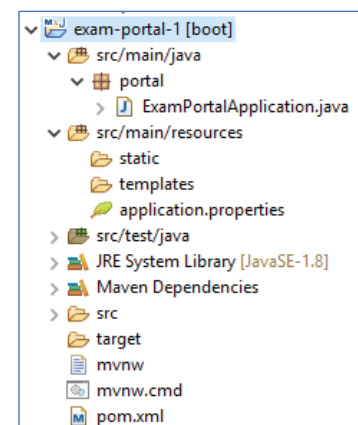
8

## Struttura del progetto creato

- *ExamPortalApplication.java* è la classe che contiene il metodo *main*: inizializza e avvia l'applicazione.
- *application.properties* è il file (all'inizio vuoto) in cui:
  - inserire le direttive di configurazione che Spring Boot non genera automaticamente (in altri file)
  - o sovrascrivere le configurazioni di default (v. oltre)
- Il file *pom.xml* (di Maven) dichiara tutte le *dipendenze* del progetto, che verranno scaricate e aggiunte al progetto automaticamente

NB: subito, o in seguito, STS può rilevare problemi in (file di) progetto, come si vede qui a destra:

- il problema è poi dettagliato nel riquadro "Problems", in basso a destra nella finestra dell'IDE, e nel file interessato



12/12/2020

Web app SpringBoot con STS

9

# La classe dell'applicazione Spring (Boot)

- inserire
- 0

```
// ExamPortalApplication.java
package portal;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ExamPortalApplication {

    public static void main(String[] args) {

        SpringApplication.run(ExamPortalApplication.class,
            args);
    }
}
```

12/12/2020

Web app SpringBoot con STS

10

## Preparare il Database

- Prima di poter avviare l'applicazione dobbiamo creare e collegare un database al nostro progetto.
- Creiamo quindi un nuovo database e un nuovo utente.
- Non serve creare alcuna tabella con i dati per ora, Spring si occuperà di creare tabelle e relazioni automaticamente utilizzando le classi che creeremo.

```
gp@ ~ $ sudo mysql -u root
Welcome to the MariaDB monitor.  Commands end with ; or \g. Your MariaDB connection id is
195... Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> rehash      -- per autocompletion! Meglio ancora in /etc/my.cnf

MariaDB [(none)]> CREATE DATABASE portal_db;
Query OK, 1 row affected (0.001 sec)

MariaDB [(none)]> CREATE USER 'portal'@'localhost' IDENTIFIED BY 'portal';  -- user e
password
Query OK, 0 rows affected (0.004 sec)

MariaDB [(none)]> GRANT ALL ON portal_db.* TO 'portal'@'localhost';  -- occhio (anche
sopra) ai 4 apici
Query OK, 0 rows affected (0.005 sec)
# Attenti a introdurre utenti per l'host "wildcard" % - significa "tutti gli host", ma
non localhost !
# portal@'localhost' e 'portal'@% sono utenti diversi (con diritti e password
possibilmente diversi)!
```

11

# Configurazione della Web App

Si modifichi in STS il file *application.properties* per dare all'applicazione accesso al DB appena creato col nuovo utente

```
# application.properties

spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/portal_db?serverTimezone=Europe/Rome&
pippo_sciuvra=ssl
spring.datasource.username=portal
spring.datasource.password=portal

server.port=3000
altra.direttiva.vedi.progetto.todisco=3000
```

N.B.: *spring.jpa.hibernate.ddl-auto* può avere questi valori:

- *none*: non apporta modifiche alla struttura del database
- *update*: aggiorna la struttura del database (tabelle) in base alle entità create
- *create*: crea ad ogni esecuzione il database (non mantiene i dati)

12/12/2020

Web app SpringBoot con STS

12

## Primo avvio (run)

```
2018-12-29 15:04:33.923 INFO 212 --- [main] portal.ExamPortalApplication : Starting ExamPortalApplication on DESKTOP-C900CSM with PID 212 (C:\Use
2018-12-29 15:04:33.926 INFO 212 --- [main] portal.ExamPortalApplication : No active profile set, falling back to default profiles: default
2018-12-29 15:04:34.862 INFO 212 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data repositories in DEFAULT mode.
2018-12-29 15:04:34.885 INFO 212 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 16ms. Found 0 repository i
2018-12-29 15:04:35.339 INFO 212 --- [main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransactionManag
2018-12-29 15:04:35.972 INFO 212 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2018-12-29 15:04:36.007 INFO 212 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2018-12-29 15:04:36.008 INFO 212 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/9.0.13
2018-12-29 15:04:36.022 INFO 212 --- [main] o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal perfor
2018-12-29 15:04:36.226 INFO 212 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2018-12-29 15:04:36.226 INFO 212 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 2231 ms
2018-12-29 15:04:36.459 INFO 212 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2018-12-29 15:04:37.108 INFO 212 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2018-12-29 15:04:37.168 INFO 212 --- [main] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [
name: default
...]
2018-12-29 15:04:37.347 INFO 212 --- [main] org.hibernate.Version : HHH000412: Hibernate Core {5.3.7.Final}
2018-12-29 15:04:37.348 INFO 212 --- [main] org.hibernate.cfg.Environment : HHH000206: hibernate.properties not found
2018-12-29 15:04:37.550 INFO 212 --- [main] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.0.4.Final}
2018-12-29 15:04:37.739 INFO 212 --- [main] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
2018-12-29 15:04:38.045 INFO 212 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2018-12-29 15:04:38.314 INFO 212 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2018-12-29 15:04:38.375 WARN 212 --- [main] aWebConfiguration$JpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database que
2018-12-29 15:04:38.655 INFO 212 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2018-12-29 15:04:38.660 INFO 212 --- [main] portal.ExamPortalApplication : Started ExamPortalApplication in 5.195 seconds (JVM running for 5.75)
2018-12-29 15:04:52.185 INFO 212 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2018-12-29 15:04:52.187 INFO 212 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2018-12-29 15:04:52.214 INFO 212 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 27 ms
```

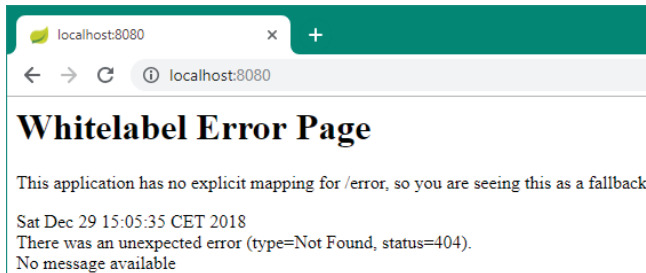
12/12/2020

Web app SpringBoot con STS

13



# Primo avvio (cont)



Spring Boot avvia automaticamente un server Tomcat per la nostra applicazione.

La porta di default è la 8080, ma è possibile modificarla in *application.properties*, con la proprietà *server.port* (nel lucido precedente è mostrata con il valore 3000).

## Entità

Il database non contiene ancora nessuna tabella.

Spring Boot all'avvio fa un'analisi delle classi del progetto e controlla che ruolo hanno nell'applicazione.

Se una classe è marcata con l'annotazione *@Entity*, Spring Boot utilizza il nome della classe e i suoi attributi per generare una tabella sul database (attraverso *JPA* e *Hibernate*).

Anche gli attributi della classe possono avere delle annotazioni, in modo da esprimere eventuali vincoli di integrità per il corrispondente attributo (colonna) della tabella



# Entità Esame

Creiamo una nuova classe *Exam* per descrivere l'entità associata agli esami.

- per questo, *Exam* è annotata *@Entity*

*Exam* ha tre attributi: un id *Long*, una descrizione e una data

- *@Id* e *@GeneratedValue* per l'attributo *id* indicano la chiave primaria per l'entità con valore auto-generato dal DBMS)
- *@NotBlank* impedisce che un attributo stringa sia vuota o *null*
- *@NotNull* invece consente la stringa vuota

NB: i metodi *getter*, *setter* e *ToString* si possono generare da menù contestuale, SOURCE-GENERATE... Comunque i loro nomi **devono** contenere i nomi gli **attributi**

In generale, capire se ci sono vincoli riguardo ai nomi delle classi

NB: sarebbe utile (ma non c'è!) un wizard che generi i template per classi entità come questa

```
// Exam.java

import java.util.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;

@Entity
public class Exam {
    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private Long id;
    @NotBlank
    private String description;
    @NotNull
    private Date date;

    public Exam() { }

    public Exam(String description, Date
        date) {
        this.description = description;
        this.date = date;
    }
    // metodi getters, setters toString()
    ...
}
```

12/12/2020

Web app SpringBoot con STS

## Entità Esame (cont)

Riavviando (STOP+RUN) l'applicazione, possiamo notare la tabella associata appena creata con le colonne che corrispondono agli attributi della classe.

L'id è chiave primaria, con valore incrementato ad ogni inserimento (auto\_increment).

```
mysql> USE portal;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_portal |
+-----+
| exam             |
+-----+
1 row in set (0.00 sec)

mysql> DESCRIBE exam;
+----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key  | Default | Extra           |
+----+-----+-----+-----+-----+-----+
| id         | bigint(20)    | NO   | PRI  | NULL    | auto_increment |
| date      | datetime      | NO   |      | NULL    |                 |
| description | varchar(255)  | YES  |      | NULL    |                 |
+----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

12/12/2020

Web app SpringBoot con STS

17

# Operazioni su tabelle

Ora che la tabella è stata creata, possiamo popolare le nostre entità.

Poi, grazie a Spring Data e ai repository JPA possiamo fare delle operazioni su queste entità senza scrivere nessuna query o metodo.

Spring grazie ai *Repository* riesce a capire dal nome del metodo che scriviamo (se è uno di quelli previsti) che operazione vogliamo fare e la implementa per noi.

N.B.: (<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories>) per dettagli su come scrivere i nomi dei metodi.

## CrudRepository (interfaccia di *org.springframework.data.repository*)

Data una entità, le operazioni più comuni sono: creazione, ricerca, aggiornamento e cancellazione. Possiamo fare tutte queste operazioni (ed altre ancora) senza scrivere una nostra implementazione.

Useremo i *CrudRepository*: CRUD sta per CreateReadUpdateDelete.

Se una nostra interfaccia eredita da *CrudRepository*, essa automaticamente eredita i metodi elencati a destra e ha a disposizione le loro implementazioni

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

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

    Optional<T> findById(ID primaryKey);      ❷

    Iterable<T> findAll();                    ❸

    long count();                             ❹

    void delete(T entity);                    ❺

    boolean existsById(ID primaryKey);        ❻

    // ... more functionality omitted.
}
```

# Repository per entità Esame

Ogni Repository è associato ad una entità, se abbiamo più entità dobbiamo creare più repository.

```
// ExamRepository.java
package portal.repository;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import portal.domain.Exam;

@Repository
public interface ExamRepository extends CrudRepository<Exam, Long> {
}
```

Per ottenere un repository per le nostre entità basta definire una nuova interfaccia, annotata con `@Repository`, che eredita da `CrudRepository`.

Il nome dell'interfaccia segue (per chiarezza) una convenzione: *ExamRepository*

`CrudRepository<>` si aspetta due tipi-parametro: il primo è la classe dell'entità, il secondo è il tipo dell'id dell'entità (nel nostro caso *Long*).

## Servizi

Ora che abbiamo il repository per gli esami, ci serve un componente che fornisca un'interfaccia per effettuare le operazioni sulla tabella esami.

Creiamo un servizio per questo compito.

Un servizio ha un riferimento al Repository e deve essere utilizzato da chi vuole accedere ai dati salvati sul database.

L'alternativa è usare direttamente il repository, accontentandosi dei metodi CRUD standard

# Classe Servizio per gli Esami

Creiamo una classe che implementa dei metodi per interagire con il repository.

L'annotazione `@Service` esplicita il ruolo che ha questa classe nella nostra applicazione.

Con questo servizio si possono:  
creare/modificare un esame, ottenere tutti gli esami o uno dato l'id, cancellare un esame.

Il servizio *ExamService* ha bisogno del repository corrispondente *ExamRepository*.

Con l'annotazione `@Autowired`, Spring

- associa un'istanza del repository all'attributo automaticamente e
- effettua la Dependency Injection

N.B.: *addExam* e *updateExam* del servizio, usano lo stesso metodo *save* del repository:

- ciò perché *save* del repository controlla l'id del parametro ricevuto e, se questo è già presente nel database modifica l'entità, altrimenti crea una nuova entità

```
// ExamService.java
package portal.service;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import portal.domain.Exam;
import portal.repository.ExamRepository;

@Service
public class ExamService {

    @Autowired
    private ExamRepository examRepository;

    public Exam addExam(Exam e) {
        return examRepository.save(e);
    }

    public List<Exam> getAllExams() {
        List<Exam> exams = new ArrayList<>();
        examRepository.findAll().forEach(exams::add);
        return exams;
    }

    public Optional<Exam> getExam(Long id) {
        return examRepository.findById(id);
    }

    public Exam updateExam(Exam e) {
        return examRepository.save(e);
    }

    public void deleteExam(Long id) {
        examRepository.deleteById(id);
    }
}
```

12/12/2020

Web app SpringBoot con STS

22

## Controller

Adesso abbiamo tutto il necessario per cominciare a costruire le REST API relative agli esami.

Abbiamo definito che struttura ha un esame e le operazioni che possiamo fare con gli esami.

Dobbiamo adesso rendere disponibili queste operazioni tramite Spring MVC.

12/12/2020

Web app SpringBoot con STS

23

# Un controller per gli esami

Creiamo una nuova classe e usiamo l'annotazione `@RestController` per dichiarare il ruolo di questa classe. Con l'annotazione `@RequestMapping` intercettiamo tutte le richieste HTTP in <http://localhost:8080/api>

Dichiariamo un riferimento al servizio `ExamService` e definiamo il primo metodo per ottenere tutti gli esami.

Con `@GetMapping` il metodo `getAllExams()` viene eseguito solo quando la richiesta HTTP a `/api/exams` è di tipo GET.

Otteniamo la lista di tutti gli esami presenti dal servizio e creiamo la risposta HTTP usando la classe `ResponseEntity`: con il metodo `.ok()` la risposta avrà `StatusCode 200 (OK)` e nel body della risposta mettiamo la lista degli esami.

Il body sarà trasformato in JSON.

```
// ExamController.java
package portal.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import portal.domain.Exam;
import portal.service.ExamService;

@RestController
@RequestMapping("/api")
public class ExamController {

    @Autowired
    private ExamService examService;

    @GetMapping("/exams")
    public ResponseEntity<List<Exam>> getAllExams() {
        List<Exam> exams = examService.getAllExams();
        return ResponseEntity.ok().body(exams);
    }
}
```

12/12/2020

Web app SpringBoot con STS

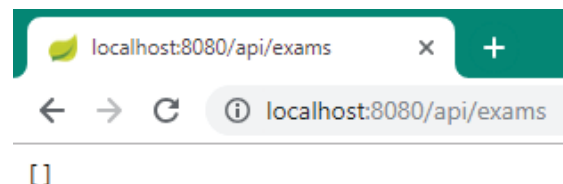
24

## La richiesta GET per tutti gli esami

Riavviando l'applicazione e usando un qualsiasi browser possiamo provare a richiedere tutti gli esami salvati.

Come ci aspettiamo otteniamo un array vuoto, ancora non ci sono esami.

Creiamo un metodo nel controller per aggiungere un esame.



12/12/2020

Web app SpringBoot con STS

25

# Richiesta POST per gli esami

```
@PostMapping("/exams")
public ResponseEntity<Exam> addExam(@Valid @RequestBody Exam e) throws URISyntaxException {
    if (e.getId() != null) {
        return ResponseEntity.badRequest().build();
    }
    Exam result = examService.addExam(e);
    return ResponseEntity.created(new URI("/api/exams/" + result.getId())).body(result);
}
```

Aggiungiamo al controller precedente il metodo `addExam()`.

Con `@PostMapping` ci mettiamo in ascolto di richieste di tipo POST (usate per creare una nuova entità).

A differenza che per GET, è richiesto il parametro: l'esame da aggiungere. Con `@RequestBody` si impone che nella richiesta ci sia un oggetto `Exam`, con `@Valid` che esso rispetti i vincoli definiti nella classe `Exam` per l'entità.

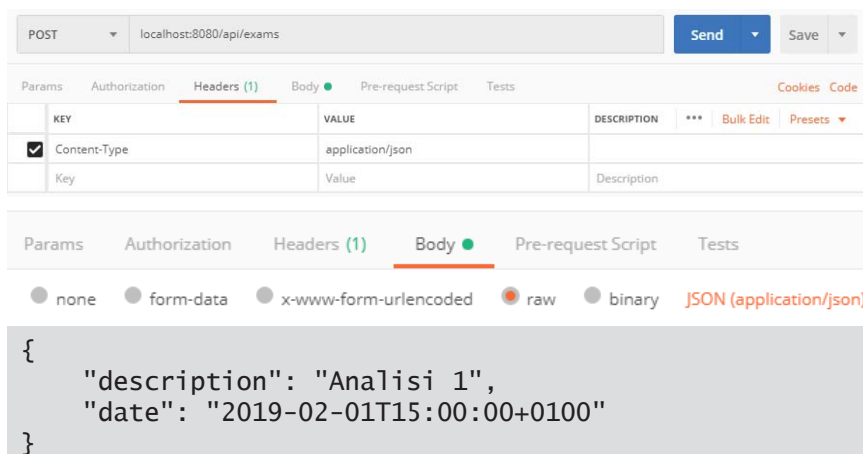
Se l'esame che si vuole aggiungere ha già un ID, restituiamo una risposta con codice 400 BAD REQUEST (un nuovo esame non deve avere già un ID).

Altrimenti aggiungiamo l'esame con il servizio, questo ci restituisce l'entità appena creata che mettiamo nella risposta con codice 201 CREATED e un riferimento all'entità (non ancora attivo).

## POST Esame

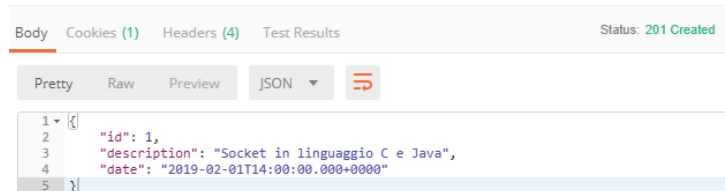
Per effettuare richieste HTTP possiamo usare il software Postman (<https://www.getpostman.com/>). Riavviamo l'applicazione e proviamo il nuovo metodo.

La nostra richiesta ha nell'intestazione il campo Content-Type settato con application/json e nel body abbiamo un oggetto JSON.

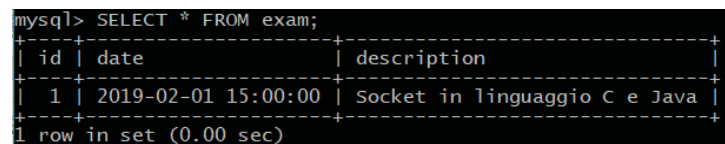


# Risposta alla POST

Cliccando su Send otteniamo la seguente risposta. Il codice è 201 Created e il body contiene l'oggetto esame appena creato.



Possiamo anche verificare sul nostro database con una semplice query.



# GET con parametro

Se siamo interessati ad ottenere uno specifico esame, possiamo creare un nuovo metodo che prende in input l'id dell'esame desiderato. Con `@PathVariable` ricaviamo dall'indirizzo l'id dell'esame che passiamo al servizio. Se l'esame esiste lo restituiamo con codice 200 OK, altrimenti restituiamo una risposta con codice 404 NOT FOUND.

```
@GetMapping("/exams/{id}")
public ResponseEntity<Exam> getExam(@PathVariable Long id) {
    Optional<Exam> exam = examService.getExam(id);
    if (exam.isPresent())
        return ResponseEntity.ok().body(exam.get());
    return ResponseEntity.notFound().build();
}
```



# GET con parametro (cont)

## Alcuni esempi di richieste GET

The screenshot shows a REST client interface with three GET requests to localhost:8080/api/exams. The first request to /api/exams/2 returns a JSON array with 3 items. The second request to /api/exams/5 returns a 404 Not Found status.

```
GET localhost:8080/api/exams/2
```

Body Cookies (1) Headers (3) Test Results

Pretty Raw Preview JSON

```
1 {
2   {
3     "id": 1,
4     "description": "Socket in linguaggio C e Java",
5     "date": "2019-02-01T14:00:00.000+0000"
6   },
7   {
8     "id": 2,
9     "description": "Thread",
10    "date": "2019-02-04T09:00:00.000+0000"
11  },
12  {
13    "id": 3,
14    "description": "Web Services",
15    "date": "2019-02-06T15:00:00.000+0000"
16  }
17 }
```

GET localhost:8080/api/exams/5

Body Cookies (1) Headers (2) Test Results

Status: 404 Not Found

12/12/2020 Web app SpringBoot con STS 30

## Modifica di un esame

Per modificare un esame già esistente usando il seguente metodo in ascolto su richieste HTTP di tipo PUT. Il metodo è molto simile al metodo per creare un nuovo esame, la differenza è nel controllo: qui l'id dell'oggetto nel body deve essere valorizzato per procedere con la modifica. Viene restituito l'oggetto modificato con il nuovo stato.

```
@PutMapping("/exams")
public ResponseEntity<Exam> updateExam(@Valid @RequestBody Exam e) {
    if (e.getId() == null) {
        return ResponseEntity.badRequest().build();
    }
    Exam result = examService.updateExam(e);
    return ResponseEntity.ok().body(result);
}
```

# PUT

L'esame con id=2 prima e dopo l'esecuzione della richiesta di modifica

The first screenshot shows a GET request to `localhost:8080/api/exams/2`. The response body is a JSON object: `{ "id": 2, "description": "Thread", "date": "2019-02-04T09:00:00.000+0000" }`.

The second screenshot shows a PUT request to `localhost:8080/api/exams`. The request body is a JSON object: `{ "id": "2", "description": "Thread e RMI in Java", "date": "2019-02-04T09:30:00+0100" }`.

The third screenshot shows a GET request to `localhost:8080/api/exams/2`. The response body is the updated JSON object: `{ "id": 2, "description": "Thread e RMI in Java", "date": "2019-02-04T08:30:00.000+0000" }`.

12/12/2020

Web app SpringBoot con STS

32

## Eliminare un esame

L'ultima operazione che rimane è la cancellazione. Richiediamo l'id dell'esame da eliminare e aspettiamo richieste HTTP di tipo DELETE.

Se esiste un esame con l'id passato, questo viene eliminato altrimenti viene restituita una risposta con codice di errore 404 NOT FOUND.

```
@DeleteMapping("/exams/{id}")
public ResponseEntity<Void> deleteExam(@PathVariable Long id) {
    if (!examService.getExam(id).isPresent()) {
        return ResponseEntity.notFound().build();
    }
    examService.deleteExam(id);
    return ResponseEntity.ok().build();
}
```

12/12/2020

Web app SpringBoot con STS

33

# Esempio di cancellazione

The screenshot illustrates a REST client interface with three panels. The top panel shows a GET request to `localhost:8080/api/exams/`. The middle panel shows a DELETE request to `localhost:8080/api/exams/3` with a status of 200 OK. The bottom panel shows the response body, which is a JSON array of three exam objects. The first object is for 'Socket in linguaggio C e Java', the second for 'Thread e RMI in Java', and the third for 'Web Services'.

```
1 [
2   {
3     "id": 1,
4     "description": "Socket in linguaggio C e Java",
5     "date": "2019-02-01T14:00:00.000+0000"
6   },
7   {
8     "id": 2,
9     "description": "Thread e RMI in Java",
10    "date": "2019-02-04T08:30:00.000+0000"
11  },
12  {
13    "id": 3,
14    "description": "Web Services",
15    "date": "2019-02-06T15:00:00.000+0000"
16  }
17 ]
```

12/12/2020

Web app SpringBoot con STS

34

## Struttura finale



12/12/2020

Web app SpringBoot con STS

35

# I risultati

Ora che abbiamo gli esami, possiamo gestire la parte dei risultati.

Un risultato è associato ad una coppia <studente, esame>, come attributi ha un voto (da valorizzare) ed eventualmente delle note.

Esiste una relazione 1 a molti tra Esame e Risultato, un esame può avere diversi risultati ma un risultato appartiene solamente ad un esame.

12/12/2020

Web app SpringBoot con STS

36

## Entità Result

Rispetto all'entità Esame, serve q  
Abbiamo un attributo exam anno  
@ManyToOne , annotazione che  
relazione tra Result ed Exam da p  
(si possono utilizzare anche le an  
@OneToMany, @OneToOne per c  
relazioni).

L'attributo exam verrà tradotto ne  
dell'esame associato al risultato.

Sarà il supporto JPA a introdurre r  
le relazioni appropriate che deriva  
annotazioni riguardanti le relazio

Come detto precedentemente, de  
aggiungere il vincolo sulla coppia  
esame> poiché deve essere unica  
della tabella. Per fare ciò usiamo l  
@Table : con il parametro name i  
nome della tabella; con uniqueCo  
relativa annotazione indichiamo c  
«student, exam\_id» deve essere u

N.B.: con l'annotazione @Size su s ...  
possiamo impostare la lunghezza  
massima che può avere l'attributo

```
import javax.persistence.UniqueConstraint;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

@Entity
@Table(name = "result", uniqueConstraints = { @UniqueConstraint(
    "exam_id" }) })
public class Result {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @NotNull
    private Exam exam;

    @NotBlank
    @Size(max = 16)
    private String student;

    @NotBlank
    private String mark;

    private String note;

    public Result() {
        // Costruttori
        // Getters e setters
    }
}
```

# Repository per Result

Oltre alle normali operazioni CRUD su Result, siamo interessati anche ad ottenere tutti i risultati di un particolare esame e anche tutti i risultati di uno studente.

Grazie a Spring Data, basta semplicemente definire i metodi con un opportuno nome.

Quando un metodo inizia con `findBy` seguito da una colonna (passando anche un parametro), richiamando questa funzione otteniamo la lista dei risultati che hanno nella colonna passata il parametro richiesto.

N.B.: non dobbiamo implementare alcun metodo.

```
// ResultRepository.java
package portal.repository;

import java.util.List;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

import portal.domain.Result;

@Repository
public interface ResultRepository extends CrudRepository<Result, Long> {
    List<Result> findByExamId(Long id);

    List<Result> findByStudent(String student);
}
```

# Servizio per Result

Molto simile a quello per gli esami, aggiungiamo due metodi per richiamare le query aggiunte al repository standard.

```
// ResultService.java
package portal.service;

import java.util.ArrayList;

@Service
public class ResultService {

    @Autowired
    private ResultRepository resultRepository;

    public Result addResult(Result r) {
        return resultRepository.save(r);
    }

    public List<Result> getAllResults() {
        List<Result> results = new ArrayList<>();
        resultRepository.findAll().forEach(results::add);
        return results;
    }

    public List<Result> getExamResults(Long examId) {
        List<Result> results = new ArrayList<>();
        resultRepository.findByExamId(examId).forEach(results::add);
        return results;
    }

    public List<Result> getStudentResults(String student) {
        List<Result> results = new ArrayList<>();
        resultRepository.findByStudent(student).forEach(results::add);
        return results;
    }

    public Optional<Result> getResult(Long id) {
        return resultRepository.findById(id);
    }

    public Result updateResult(Result r) {
        return resultRepository.save(r);
    }

    public void deleteResult(Long id) {
        resultRepository.deleteById(id);
    }
}
```

# Result Controller

```
// ResultController.java
package portal.controller;

import java.net.URI;

@RestController
@RequestMapping("/api")
public class ResultController {

    @Autowired
    private ResultService resultService;

    @Autowired
    private ExamService examService;

    @GetMapping("/results")
    public ResponseEntity<List<Result>> getAllResults() {
        List<Result> results = resultService.getAllResults();
        return ResponseEntity.ok().body(results);
    }

    @GetMapping("/exams/{exam}/results")
    public ResponseEntity<List<Result>> getAllResults(@PathVariable Long exam) {
        List<Result> results = resultService.getExamResults(exam);
        return ResponseEntity.ok().body(results);
    }

    @GetMapping("/user/{student}/results")
    public ResponseEntity<List<Result>> getAllStudentResults(@PathVariable String student) {
        List<Result> results = resultService.getStudentResults(student);
        return ResponseEntity.ok().body(results);
    }

    @GetMapping("/results/{result}")
    public ResponseEntity<Result> getResult(@PathVariable Long result) {
        Optional<Result> res = resultService.getResult(result);
        if (res.isPresent()) {
            return ResponseEntity.ok().body(res.get());
        }
        return ResponseEntity.notFound().build();
    }

    @PostMapping("/results")
    public ResponseEntity<Result> addResult(@Valid @RequestBody Result r) throws URISyntaxException {
        if (r.getId() != null) {
            return ResponseEntity.badRequest().build();
        }
        if (!examService.getExam(r.getExam().getId()).isPresent()) {
            return ResponseEntity.notFound().build();
        }
        Result res = resultService.addResult(r);
        return ResponseEntity.created(new URI("/api/results/" + res.getId())).body(res);
    }

    @PutMapping("/results")
    public ResponseEntity<Result> updateResult(@Valid @RequestBody Result r) {
        if (r.getId() == null) {
            return ResponseEntity.badRequest().build();
        }
        if (!examService.getExam(r.getExam().getId()).isPresent()) {
            return ResponseEntity.notFound().build();
        }
        Result res = resultService.updateResult(r);
        return ResponseEntity.ok().body(res);
    }

    @DeleteMapping("/results/{result}")
    public ResponseEntity<Void> deleteResult(@PathVariable Long result) {
        if (!resultService.getResult(result).isPresent()) {
            return ResponseEntity.notFound().build();
        }
        resultService.deleteResult(result);
        return ResponseEntity.ok().build();
    }
}
```

12/12/2020

Web app SpringBoot con STS

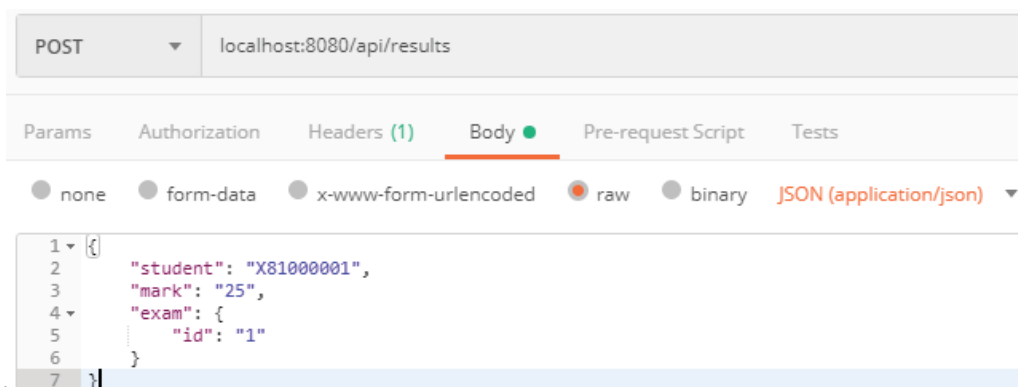
40

## Aggiungere un risultato

Esempio: lo studente con matricola X81000001 ha riportato un voto di 25 nell'esame avente id 1

La richiesta HTTP per aggiungere questo risultato è la seguente

N.B.: le note non sono obbligatorie, possono essere omesse e avranno valore null



12/12/2020

Web app SpringBoot con STS

41

# Tutti i risultati di tutti gli esami

Una GET su /results ci restituisce la lista di tutti i risultati di tutti gli esami

```
GET localhost:8080/api/results

Pretty Raw Preview JSON

1-45
[
  {
    "id": 1,
    "exam": {
      "id": 1,
      "description": "Socket in linguaggio C e Java",
      "date": "2019-02-01T14:00:00.000+0000"
    },
    "student": "X81000001",
    "mark": "25",
    "note": null
  },
  {
    "id": 2,
    "exam": {
      "id": 2,
      "description": "Thread e RMI in Java",
      "date": "2019-02-04T08:30:00.000+0000"
    },
    "student": "X81000001",
    "mark": "30L",
    "note": null
  },
  {
    "id": 3,
    "exam": {
      "id": 1,
      "description": "Socket in linguaggio C e Java",
      "date": "2019-02-01T14:00:00.000+0000"
    },
    "student": "X81000002",
    "mark": "30L",
    "note": null
  },
  {
    "id": 4,
    "exam": {
      "id": 1,
      "description": "Socket in linguaggio C e Java",
      "date": "2019-02-01T14:00:00.000+0000"
    },
    "student": "X81000003",
    "mark": "18",
    "note": null
  }
]
```

12/12/2020

Web app SpringBoot con STS

42

# Tutti i risultati di un esame

Esempio: tutti i risultati dell'esame con id 2

```
GET localhost:8080/api/exams/2/results

Pretty Raw Preview JSON

1-24
[
  {
    "id": 2,
    "exam": {
      "id": 2,
      "description": "Thread e RMI in Java",
      "date": "2019-02-04T08:30:00.000+0000"
    },
    "student": "X81000001",
    "mark": "30L",
    "note": null
  },
  {
    "id": 5,
    "exam": {
      "id": 2,
      "description": "Thread e RMI in Java",
      "date": "2019-02-04T08:30:00.000+0000"
    },
    "student": "X81000004",
    "mark": "18",
    "note": null
  }
]
```

12/12/2020

Web app SpringBoot con STS

43



# Tutti i risultati di uno studente

Esempio: tutti i risultati dello studente con matricola X81000001

```
GET localhost:8080/api/user/X81000001/results

Pretty Raw Preview JSON

[
  {
    "id": 1,
    "exam": {
      "id": 1,
      "description": "Socket in linguaggio C e Java",
      "date": "2019-02-01T14:00:00.000+0000"
    },
    "student": "X81000001",
    "mark": "25",
    "note": null
  },
  {
    "id": 2,
    "exam": {
      "id": 2,
      "description": "Thread e RMI in Java",
      "date": "2019-02-04T08:30:00.000+0000"
    },
    "student": "X81000001",
    "mark": "30",
    "note": null
  }
]
```

## Sicurezza

Allo stato attuale tutti possono fare tutto. Non c'è sicurezza.

Implementiamo un sistema di autenticazione e autorizzazione basato sui ruoli (Role-Based Access Control).

Useremo il modulo Spring Security e JWT (<https://jwt.io>).

Tutte le operazioni saranno protette, le richieste non autorizzate saranno negate.

# Aggiungere le nuove dipendenze

Apriamo il file pom.xml e aggiungiamo queste due dipendenze dentro il tag dependencies.

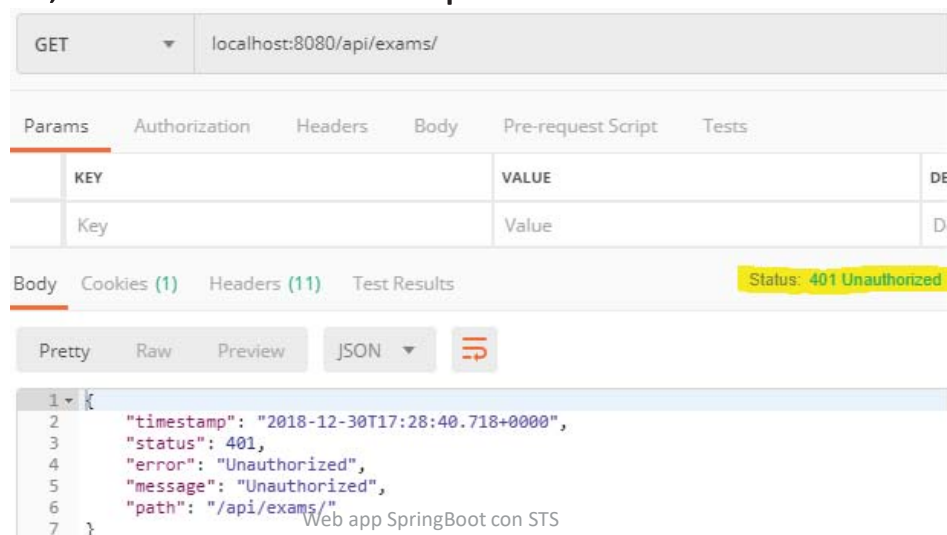
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
  <version>0.9.0</version>
</dependency>
```

Appena salviamo il file, il nostro IDE scarica e aggiunge le nuove dipendenze al nostro progetto automaticamente.

## 401 Unauthorized

Solo per aver inserito la dipendenza security, se riavviamo l'applicazione e proviamo a fare una qualsiasi richiesta, otteniamo una risposta con codice di errore 401.



# RBAC

Creiamo due ruoli per la nostra applicazione: ADMIN e USER.

Un ADMIN ha accesso totale a tutte le operazioni.

Un USER può:

- visualizzare tutti gli esami
- visualizzare solo i propri risultati

## Ruoli

Creiamo un enum per descrivere i nostri ruoli e creiamo una nuova Entity

```
// RoleName.java
package portal.domain;

public enum RoleName {
    ROLE_USER, ROLE_ADMIN
}

// Role.java
package portal.domain;

import javax.persistence.Column;

@Entity
@Table(name = "roles")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Enumerated(EnumType.STRING)
    @NaturalId
    @Column(length = 60)
    private RoleName name;

    public Role() {
    }

    public Role(RoleName name) {
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public RoleName getName() {
        return name;
    }

    public void setName(RoleName name) {
        this.name = name;
    }
}
```

# Utente

L'entità Utente ha i seguenti campi:

- un id univoco
- una stringa per memorizzare il nome e il cognome
- username
- email
- password
- un insieme di ruoli

L'username e l'email di un utente hanno il vincolo di unicità nella tabella.

Tra Role e User esiste una relazione molti a molti (@ManyToMany) e usiamo un Set<Role> per memorizzare i ruoli di un utente.

Con @JoinTable creiamo una tabella per la relazione molti a molti dove memorizziamo gli id degli utenti e dei ruoli per definire l'appartenenza di un utente ad un ruolo.

L'annotazione @Email impone il formato all'attributo email (deve essere un indirizzo valido).

```
// User.java
package portal.domain;

import java.util.HashSet;

@Entity
@Table(name = "users", uniqueConstraints = { @UniqueConstraint(columnNames = { "username" }),
                                             @UniqueConstraint(columnNames = { "email" }) })
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank
    @Size(min = 3, max = 50)
    private String name;

    @NotBlank
    @Size(min = 3, max = 50)
    private String username;

    @NaturalId
    @NotBlank
    @Size(max = 50)
    @Email
    private String email;

    @NotBlank
    @Size(min = 6, max = 100)
    private String password;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "user_roles", joinColumns = @JoinColumn(name = "user_id"), inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles = new HashSet<>();

    public User() {
    }

    public User(String name, String username, String email, String password) {
        this.name = name;
        this.username = username;
        this.email = email;
        this.password = password;
    }
}
```

12/12/2020

Web app SpringBoot con STS

50

## Creiamo i Repository per utenti e ruoli

Aggiungiamo dei metodi che ci serviranno per controllare se durante la registrazione, un username o un indirizzo email è stato già registrato.

```
// RoleRepository.java
package portal.repository;

import java.util.Optional;

@Repository
public interface RoleRepository extends CrudRepository<Role, Long> {
    Optional<Role> findByName(RoleName roleName);
}

// UserRepository.java
package portal.repository;

import java.util.Optional;

@Repository
public interface UserRepository extends CrudRepository<User, Long> {
    Optional<User> findByUsername(String username);

    Boolean existsByUsername(String username);

    Boolean existsByEmail(String email);
}
```

12/12/2020

Web app SpringBoot con STS

51

# UserPrincipal

È una classe che implementa l'interfaccia UserDetails offerta da Spring. Memorizza le informazioni dell'utente che saranno incapsulate in oggetti Authentication. Usiamo questa classe per memorizzare ulteriori informazioni non inizialmente previste (id, nome ed email).

12/12/2020

Web app SpringBoot con STS

52

## UserPrincipal (cont)

```
// UserPrincipal.java
package portal.service;

import java.util.Collection;

public class UserPrincipal implements UserDetails {
    private static final long serialVersionUID = 1L;
    private Long id;
    private String name;
    private String username;
    private String email;
    @JsonIgnore
    private String password;
    private Collection<? extends GrantedAuthority> authorities;

    public UserPrincipal(Long id, String name, String username, String email, String password,
        Collection<? extends GrantedAuthority> authorities) {
        this.id = id;
        this.name = name;
        this.username = username;
        this.email = email;
        this.password = password;
        this.authorities = authorities;
    }

    public static UserPrincipal build(User user) {
        List<GrantedAuthority> authorities = user.getRoles().stream()
            .map(role -> new SimpleGrantedAuthority(role.getName().name()))
            .collect(Collectors.toList());

        return new UserPrincipal(user.getId(), user.getName(), user.getUsername(), user.getEmail(), user.getPassword(),
            authorities);
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }

    @Override
    public String getUsername() {
        return username;
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }
}
```

```
@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return true;
}

@Override
public boolean equals(Object o) {
    if (this == o)
        return true;
    if (o == null || getClass() != o.getClass())
        return false;

    UserPrincipal user = (UserPrincipal) o;
    return Objects.equals(id, user.id);
}
```

Web app SpringBoot con STS

53

# UserDetailsServiceImpl

Una classe che implementa l'interfaccia UserDetailsService.

Dato un username, restituisce le informazioni dell'utente o genera un'eccezione se l'utente non è stato trovato.

```
// UserDetailsServiceImpl.java
package portal.service;

import org.springframework.beans.factory.annotation.Autowired;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    UserRepository userRepository;

    @Override
    @Transactional
    public User loadUserByUsername(String username) throws UsernameNotFoundException {

        User user = userRepository.findByUsername(username).orElseThrow(
            () -> new UsernameNotFoundException("User Not Found with -> username or email : " + username));

        return UserPrincipal.build(user);
    }
}
```

12/12/2020 Web app SpringBoot con STS 54

# JwtProvider

JwtProvider è una classe di servizio. Può generare un token, validarne uno o restituire l'username associato ad un token.

```
// JwtProvider.java
package portal.security.jwt;

import java.util.Date;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.core.Authentication;
import org.springframework.stereotype.Component;

import io.jsonwebtoken.ExpiredJwtException;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.MalformedJwtException;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.SignatureException;
import io.jsonwebtoken.UnsupportedJwtException;
import portal.service.UserPrincipal;

@Component
public class JwtProvider {

    private static final Logger logger = LoggerFactory.getLogger(JwtProvider.class);
    private final String jwtSecret = "JwtExamPortalAppSecret";
    private final int jwtExpiration = 86400;

    public String generateJwtToken(Authentication authentication) {
        UserPrincipal userPrincipal = (UserPrincipal) authentication.getPrincipal();
        return Jwts.builder().setSubject((userPrincipal.getUsername())).setIssuedAt(new Date())
            .setExpiration(new Date((new Date()).getTime() + jwtExpiration * 1000))
            .signWith(SignatureAlgorithm.HS512, jwtSecret).compact();
    }

    public boolean validateJwtToken(String authToken) {
        try {
            Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(authToken);
            return true;
        } catch (SignatureException e) {
            logger.error("Invalid JWT signature -> Message: {} ", e);
        } catch (MalformedJwtException e) {
            logger.error("Invalid JWT token -> Message: {} ", e);
        } catch (ExpiredJwtException e) {
            logger.error("Expired JWT token -> Message: {} ", e);
        } catch (UnsupportedJwtException e) {
            logger.error("Unsupported JWT token -> Message: {} ", e);
        } catch (IllegalArgumentException e) {
            logger.error("JWT claims string is empty -> Message: {} ", e);
        }
        return false;
    }

    public String getUsernameFromJwtToken(String token) {
        return Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(token).getBody().getSubject();
    }
}
```

12/12/2020 Web app SpringBoot con STS 55

# JwtAuthEntryPoint

È una classe che implementa l'interfaccia `AuthenticationEntryPoint`. Interviene quando avviene una richiesta che non è autorizzata.

```
// JwtAuthEntryPoint.java
package portal.security.jwt;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

@Component
public class JwtAuthEntryPoint implements AuthenticationEntryPoint {

    private static final Logger logger = LoggerFactory.getLogger(JwtAuthEntryPoint.class);

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException e)
        throws IOException {

        logger.error("Unauthorized error. Message - {}", e.getMessage());
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Error -> Unauthorized");
    }
}
```

12/12/2020

Web app SpringBoot con STS

56

# JwtAuthTokenFilter

È una classe che implementa l'interfaccia `OncePerRequestFilter`. Viene eseguita ad ogni richiesta HTTP. Il metodo `doFilterInternal`:

- Preleva il token dall'intestazione
- Valida il token
- Ottiene l'username dal token validato
- Carica le informazioni dell'utente e costruisce un oggetto `Authentication`
- Assegna l'oggetto appena creato al `Security Context` di Spring

```
// JwtAuthTokenFilter.java
package portal.security.jwt;

import java.io.IOException;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import org.springframework.stereotype.Component;

@Component
public class JwtAuthTokenFilter extends OncePerRequestFilter {

    private static final Logger logger = LoggerFactory.getLogger(JwtAuthTokenFilter.class);

    @Autowired
    private JwtProvider tokenProvider;

    @Autowired
    private UserDetailsServiceImpl userDetailsServiceImpl;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {

        try {
            String jwt = getJwt(request);
            if (jwt != null && tokenProvider.validateJwtToken(jwt)) {
                String username = tokenProvider.getUserNameFromJwtToken(jwt);
                UserDetails userDetails = userDetailsServiceImpl.loadUserByUsername(username);
                UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(
                    userDetails, null, userDetails.getAuthorities());
                authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        } catch (Exception e) {
            logger.error("Can NOT set user authentication -> Message: {}", e);
        }
        filterChain.doFilter(request, response);
    }

    private String getJwt(HttpServletRequest request) {
        String authHeader = request.getHeader("Authorization");
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            return authHeader.replace("Bearer ", "");
        }
        return null;
    }
}
```

12/12/2020

Web app SpringBoot con STS

57



# WebSecurityConfig

Classe di configurazione.

- Setta l'encoder delle password degli utenti (non vengono salvate password in chiaro)
- Applica i filtri creati precedentemente
- Ad utenti non autenticati permette di utilizzare solo /api/auth per effettuare l'accesso o la registrazione

12/12/2020

```
// WebSecurityConfig.java
package portal.security;

import org.springframework.beans.factory.annotation.Autowired;

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    UserDetailsServiceImpl userDetailsService;

    @Autowired
    private JwtAuthEntryPoint unauthorizedHandler;

    @Bean
    public JwtAuthTokenFilter authenticationJwtTokenFilter() {
        return new JwtAuthTokenFilter();
    }

    @Override
    public void configure(AuthenticationManagerBuilder authenticationManagerBuilder) throws Exception {
        authenticationManagerBuilder.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
    }

    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors().and().csrf().disable().authorizeRequests().antMatchers("/api/auth/**").permitAll().anyRequest()
            .authenticated().and().exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

        http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);
    }
}
```

## Controller per l'autenticazione

Prima di costruire il controller per l'autenticazione, creiamo delle classi che definiscono come deve essere una richiesta di login/registrazione e come deve essere la risposta

# Struttura richieste di login e registrazione

```
// SignUpForm.java
package portal.message.request;

import java.util.Set;

import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

public class SignUpForm {

    @NotBlank
    @Size(min = 3, max = 50)
    private String name;

    @NotBlank
    @Size(min = 3, max = 50)
    private String username;

    @NotBlank
    @Size(max = 60)
    @Email
    private String email;

    private Set<String> role;

    @NotBlank
    @Size(min = 6, max = 40)
    private String password;

    // Getter e Setter
}
```

12/12/2020

```
// LoginForm.java
package portal.message.request;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

public class LoginForm {

    @NotBlank
    @Size(min = 3, max = 60)
    private String username;

    @NotBlank
    @Size(min = 6, max = 40)
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

Web app SpringBoot con STS

60

# Struttura delle risposte

```
// JwtResponse.java
package portal.message.response;

import java.util.Collection;

import org.springframework.security.core.GrantedAuthority;

public class JwtResponse {

    private String token;
    private String type = "Bearer";
    private String username;
    private Collection<? extends GrantedAuthority> authorities;

    public JwtResponse(String accessToken, String username, Collection<? extends GrantedAuthority> authorities) {
        this.token = accessToken;
        this.username = username;
        this.authorities = authorities;
    }

    // Getter e Setter
}
```

```
// ResponseMessage.java
package portal.message.response;

public class ResponseMessage {

    private String message;

    public ResponseMessage(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

12/12/2020

Web app SpringBoot con STS

61

# AuthController - Login

La parte più importante: il controller dell'autenticazione.

Per il login ci mettiamo in ascolto su `/api/auth/signin` per richieste HTTP di tipo POST. Richiediamo nel body della richiesta un oggetto di `LoginForm` (definito precedentemente) che contiene l'username e la password dell'utente che vuole effettuare il login. Se il login va a buon fine autenticiamo l'utente e generiamo un token JWT che restituiamo nella risposta usando `JwtResponse`. Questo token identifica l'utente ed è necessario in ogni sua successiva richiesta HTTP.

```
// AuthController.java
package portal.controller;

import java.util.HashSet;

@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/auth")
public class AuthController {

    @Autowired
    AuthenticationManager authenticationManager;

    @Autowired
    UserRepository userRepository;

    @Autowired
    RoleRepository roleRepository;

    @Autowired
    PasswordEncoder encoder;

    @Autowired
    JwtProvider jwtProvider;

    @PostMapping("/signin")
    public ResponseEntity<?> authenticateUser(@Valid @RequestBody LoginForm loginRequest) {

        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(loginRequest.getUsername(), loginRequest.getPassword()));

        SecurityContextHolder.getContext().setAuthentication(authentication);

        String jwt = jwtProvider.generateJwtToken(authentication);
        UserDetails userDetails = (UserDetails) authentication.getPrincipal();

        return ResponseEntity.ok(new JwtResponse(jwt, userDetails.getUsername(), userDetails.getAuthorities()));
    }
}
```

12/12/2020

Web app SpringBoot con STS

62

# AuthController - Registrazione

Per la registrazione di un nuovo utente aspettiamo richieste HTTP di tipo POST su `/api/auth/signup`.

Richiediamo nel body un oggetto di tipo `SignUpForm` e controlliamo se esiste già un utente registrato con l'username o l'email passato nel form di registrazione. Se passa questo controllo, registriamo l'utente assegnando il ruolo USER. Infine viene restituita una risposta con l'esito della registrazione.

```
@PostMapping("/signup")
public ResponseEntity<?> registerUser(@Valid @RequestBody SignUpForm signUpRequest) {
    if (userRepository.existsByUsername(signUpRequest.getUsername())) {
        return new ResponseEntity<>(new ResponseMessage("Fail -> Username is already taken!"),
            HttpStatus.BAD_REQUEST);
    }

    if (userRepository.existsByEmail(signUpRequest.getEmail())) {
        return new ResponseEntity<>(new ResponseMessage("Fail -> Email is already in use!"),
            HttpStatus.BAD_REQUEST);
    }

    User user = new User(signUpRequest.getName(), signUpRequest.getUsername(), signUpRequest.getEmail(),
        encoder.encode(signUpRequest.getPassword()));

    Set<Role> roles = new HashSet<>();

    Role userRole = roleRepository.findByName(RoleName.ROLE_USER)
        .orElseThrow(() -> new RuntimeException("Fail! -> Cause: User Role not find."));
    roles.add(userRole);

    user.setRoles(roles);
    userRepository.save(user);

    return new ResponseEntity<>(new ResponseMessage("User registered successfully!"), HttpStatus.OK);
}
```

12/12/2020

Web app SpringBoot con STS

63

# Inizializzare il database

Ora che abbiamo tutto pronto, dobbiamo preparare il nostro database. Dobbiamo inserire i ruoli e aggiungere un utente amministratore. Tutti i nuovi utenti avranno il ruolo USER.

Possiamo fare questo creando un nuovo file chiamato «data.sql». Se nel classpath del progetto è presente questo file, Spring esegue le query del file all'avvio dell'applicazione. Aggiungiamo una nuova configurazione nel file application.properties per far importare il file data.sql

```
# application.properties
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/portal?serverTimezone=Europe/Rome
spring.datasource.username=portaluser
spring.datasource.password=portalpassword
spring.datasource.initialization-mode=always

-- data.sql

-- Inserisci i ruoli se non esistono
INSERT IGNORE INTO roles(name) VALUES ('ROLE_USER'),('ROLE_ADMIN');

-- Se non presente inserisci un utente amministratore e assegna entrambi i ruoli
INSERT IGNORE INTO users(name, username, password, email) VALUES
('Admin', 'admin', '$2a$10$CPClv9ShoEM3Fc2P32NkLuXcau2jUN8k2g515hkB0qgMABc4.1hy.', 'admin@email.com');
INSERT IGNORE INTO user_roles(user_id, role_id) VALUES (1,1),(1,2);
```

```
mysql> SELECT U.username, R.name FROM users U, roles R, user_roles UR WHERE UR.user_id=U.id AND UR.role_id=R.id;
+-----+-----+
| username | name |
+-----+-----+
| admin    | ROLE_USER |
| admin    | ROLE_ADMIN |
+-----+-----+
```

## Azioni protette

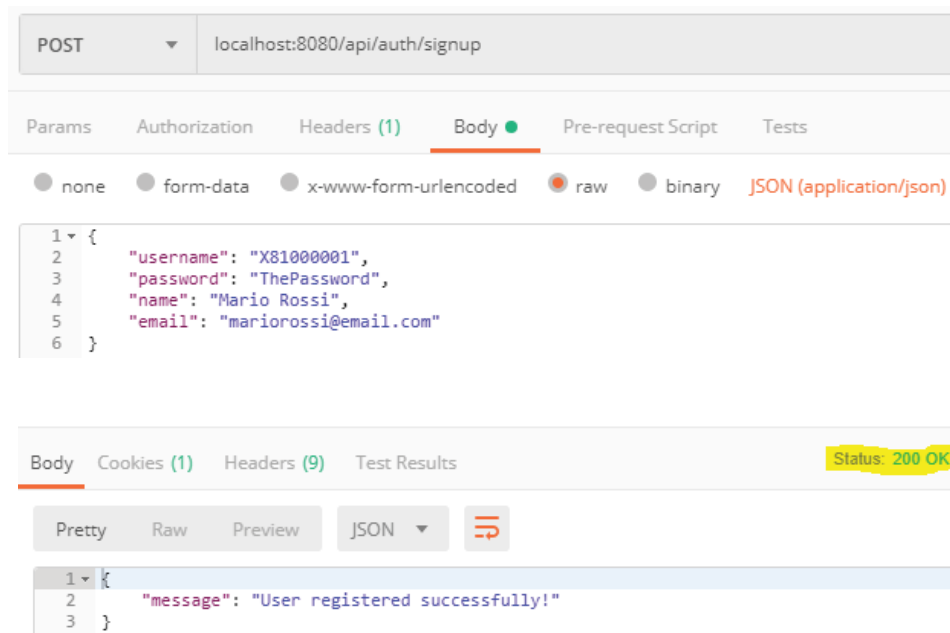
Ora che abbiamo i ruoli, possiamo permettere o vietare l'esecuzione dei metodi in funzione del ruolo dell'utente che richiede l'azione.

Per fare questo usiamo l'annotazione `@PreAuthorize`

Per esempio: vogliamo far aggiungere esami solo agli amministratori, aggiungiamo questa annotazione al metodo del controller per gli esami.

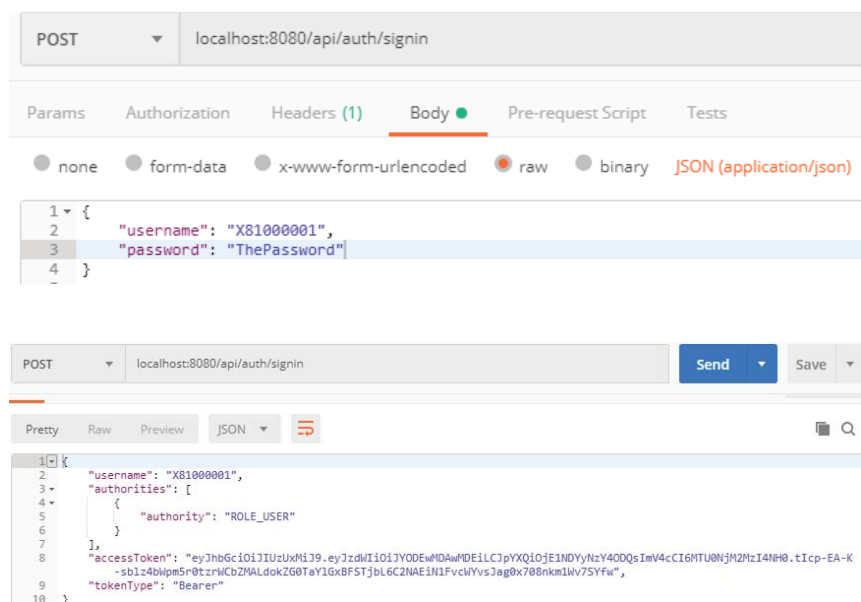
```
@PostMapping("/exams")
@PreAuthorize("hasRole('ADMIN')")
public ResponseEntity<Exam> addExam(@Valid @RequestBody Exam e) throws URISyntaxException {
    if (e.getId() != null) {
        return ResponseEntity.badRequest().build();
    }
    Exam result = examService.addExam(e);
    return ResponseEntity.created(new URI("/api/exams/" + result.getId())).body(result);
}
```

# Registrazione di un utente



66

## Login di un utente



67

# GET su esami con token

Con il login viene restituito il token dell'utente appena connesso. Questo deve far parte dell'intestazione di ogni richiesta per passare il controllo di accesso.

Se il token non è presente viene restituito una risposta con codice 401 Unauthorized.

The left screenshot shows a GET request to `localhost:8080/api/exams` with no headers. The response is a JSON object with status 401, indicating unauthorized access.

```
1 {
2   "timestamp": "2018-12-31T17:26:10.984+0000",
3   "status": 401,
4   "error": "Unauthorized",
5   "message": "Error -> Unauthorized",
6   "path": "/api/exams"
7 }
```

The right screenshot shows the same GET request but with an `Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJYOD...` header. The response is a JSON array of two exam objects with status 200.

```
1 [
2   {
3     "id": 1,
4     "description": "Socket in linguaggio C e Java",
5     "date": "2019-02-01T14:00:00.000+0000"
6   },
7   {
8     "id": 2,
9     "description": "Thread e RMI in Java",
10    "date": "2019-02-04T08:30:00.000+0000"
11  }
12 ]
```

12/12/2020

Web app SpringBoot con STS

68

# POST su esame

Se lo stesso utente prova a creare un nuovo esame, riceve una risposta con codice 403 perché non è autorizzato a creare nuovi esami.

È importante aggiungere l'annotazione `@PreAuthorize` ai metodi con risorse e operazioni sensibili.

The screenshot shows a POST request to `localhost:8080/api/exams` with headers `Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJYOD...` and `Content-Type: application/json`. The response is a JSON object with status 403, indicating forbidden access.

```
1 {
2   "timestamp": "2018-12-31T17:29:10.195+0000",
3   "status": 403,
4   "error": "Forbidden",
5   "message": "Forbidden",
6   "path": "/api/exams"
7 }
```

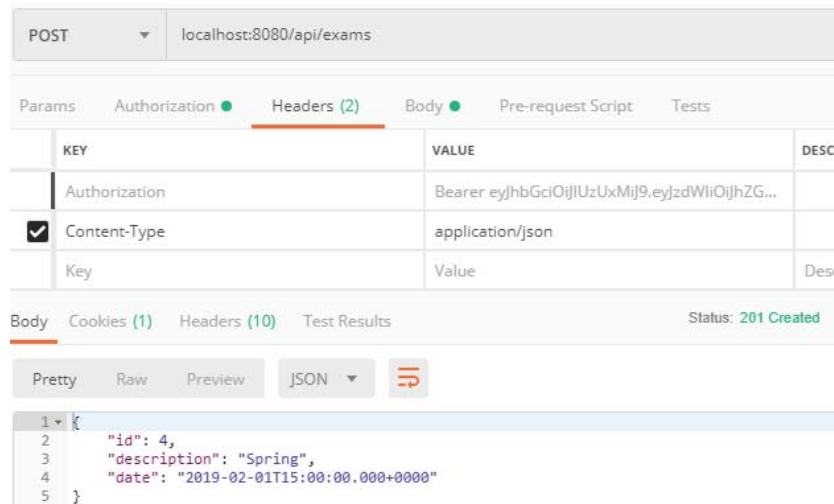
12/12/2020

Web app SpringBoot con STS

69

# POST su esame (cont)

Se il token appartiene ad un utente con il ruolo amministratore, l'operazione viene eseguita normalmente.



12/12/2020

Web app SpringBoot con STS

70

## Risultati dell'utente connesso

Adesso possiamo aggiungere un nuovo metodo al controller dei risultati per far restituire i risultati dell'utente attualmente connesso.

Ricaviamo l'utente attualmente connesso che fa la richiesta utilizzando il SecurityContext. Il SecurityContext possiede l'oggetto Authentication che contiene a sua volta i dati dell'utente. Prendiamo l'username e attraverso il servizio prendiamo i suoi risultati che restituiamo nella risposta HTTP.

```
@GetMapping("/user/results")
public ResponseEntity<List<Result>> getStudentResults() {
    String user = SecurityContextHolder.getContext().getAuthentication().getName();
    List<Result> results = resultService.getStudentResults(user);
    return ResponseEntity.ok().body(results);
}
```

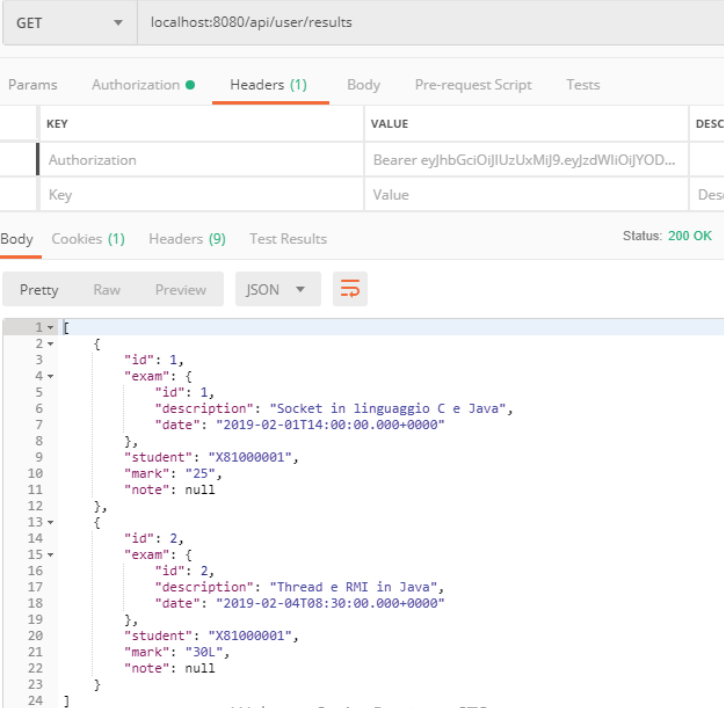
12/12/2020

Web app SpringBoot con STS

71



# Risultati dell'utente connesso (cont)



GET localhost:8080/api/user/results

Params Authorization Headers (1) Body Pre-request Script Tests

KEY	VALUE	DESC
Authorization	Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWliOiJYOD...	
Key	Value	Desc

Body Cookies (1) Headers (9) Test Results Status: 200 OK

Pretty Raw Preview JSON

```
1 [
2   {
3     "id": 1,
4     "exam": {
5       "id": 1,
6       "description": "Socket in linguaggio C e Java",
7       "date": "2019-02-01T14:00:00.000+0000"
8     },
9     "student": "X81000001",
10    "mark": "25",
11    "note": null
12  },
13  {
14    "id": 2,
15    "exam": {
16      "id": 2,
17      "description": "Thread e RMI in Java",
18      "date": "2019-02-04T08:30:00.000+0000"
19    },
20    "student": "X81000001",
21    "mark": "30L",
22    "note": null
23  }
24 ]
```

12/12/2020

Web app SpringBoot con STS

72

## Build

Per generare il file JAR all'interno della cartella del progetto:

\$ mvnw clean package

Al termine dell'esecuzione dentro la cartella target troveremo il file .jar pronto per essere eseguito

12/12/2020

Web app SpringBoot con STS

73

# Conclusioni

Adesso ci sono tutti gli strumenti necessari per completare il back-end di questa applicazione. Abbiamo visto: come creare nuove entità; come usare i repository CRUD per le principali query e come aggiungerne altre dichiarando solo un metodo; come rendere disponibili azioni su queste entità usando richieste HTTP e servizi.

Infine abbiamo aggiunto sicurezza all'applicazione implementando un sistema RBAC che fa uso di token JWT.

Adesso non rimane altro da fare che sviluppare il front-end (usando Angular o React per esempio).