

# 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 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 Spring Tool Suite)
- Postman (per testing, facoltativo)

# Descrizione progetto

Utilizziamo 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:

- Spring Data e JDBC: per 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

Spring Initializr ( <https://start.spring.io> ) è uno strumento per creare progetti Spring Boot.

Inserendo 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 esclusivamente dentro STS (che è in grado di interfacciarsi con Initializr)

The screenshot shows the Spring Initializr web interface. At the top, it says "SPRING INITIALIZR bootstrap your application". Below this, there are fields to "Generate a"  "with"  "and Spring Boot" . The "Project Metadata" section has "Artifact coordinates" with "Group"  and "Artifact" . The "Dependencies" section has a search bar with "Web, Security, JPA, Actuator, Devtools..." and a list of "Selected Dependencies" including "Web", "JPA", and "MySQL". A green button at the bottom right says "Generate Project alt + ⌘".

SPRING INITIALIZR bootstrap your application

Generate a  with   
and Spring Boot

Project Metadata

Artifact coordinates

Group

Artifact

Dependencies

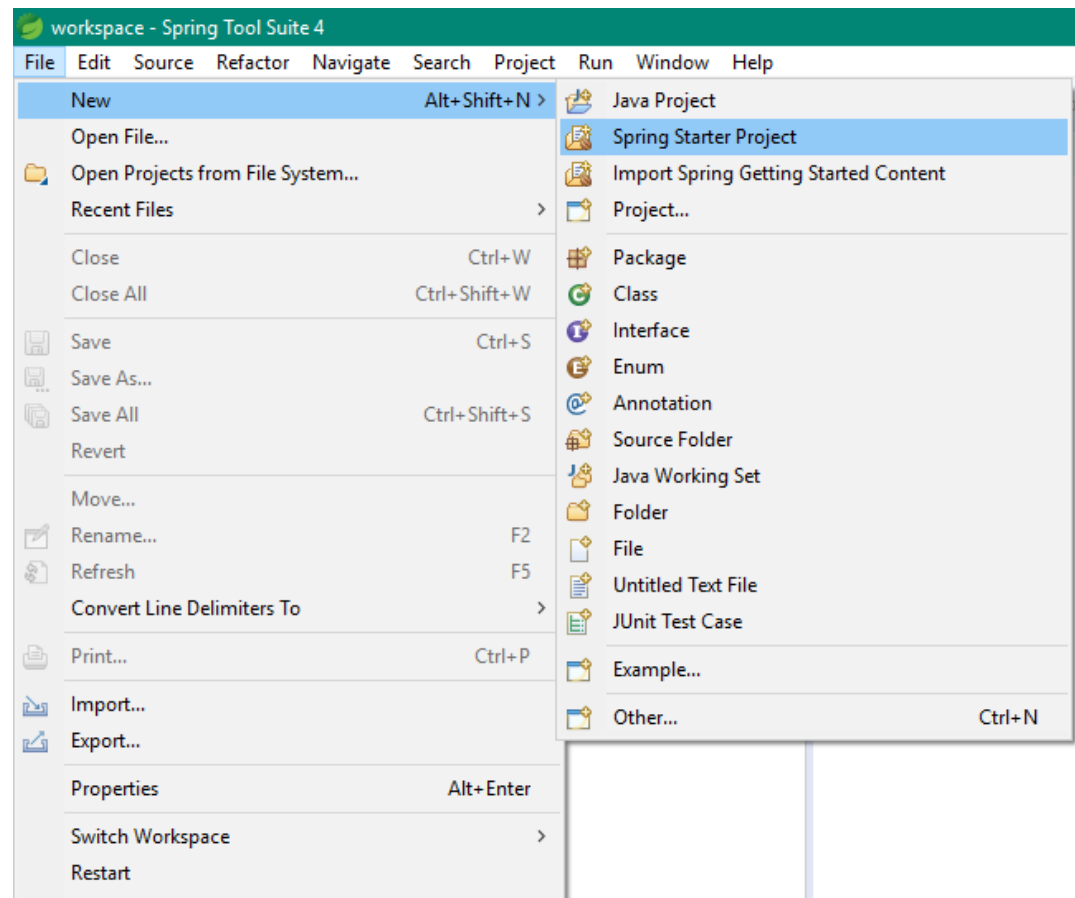
Add Spring Boot Starters and dependencies to your application

Search for dependencies

Selected Dependencies

# Creare un progetto usando STS

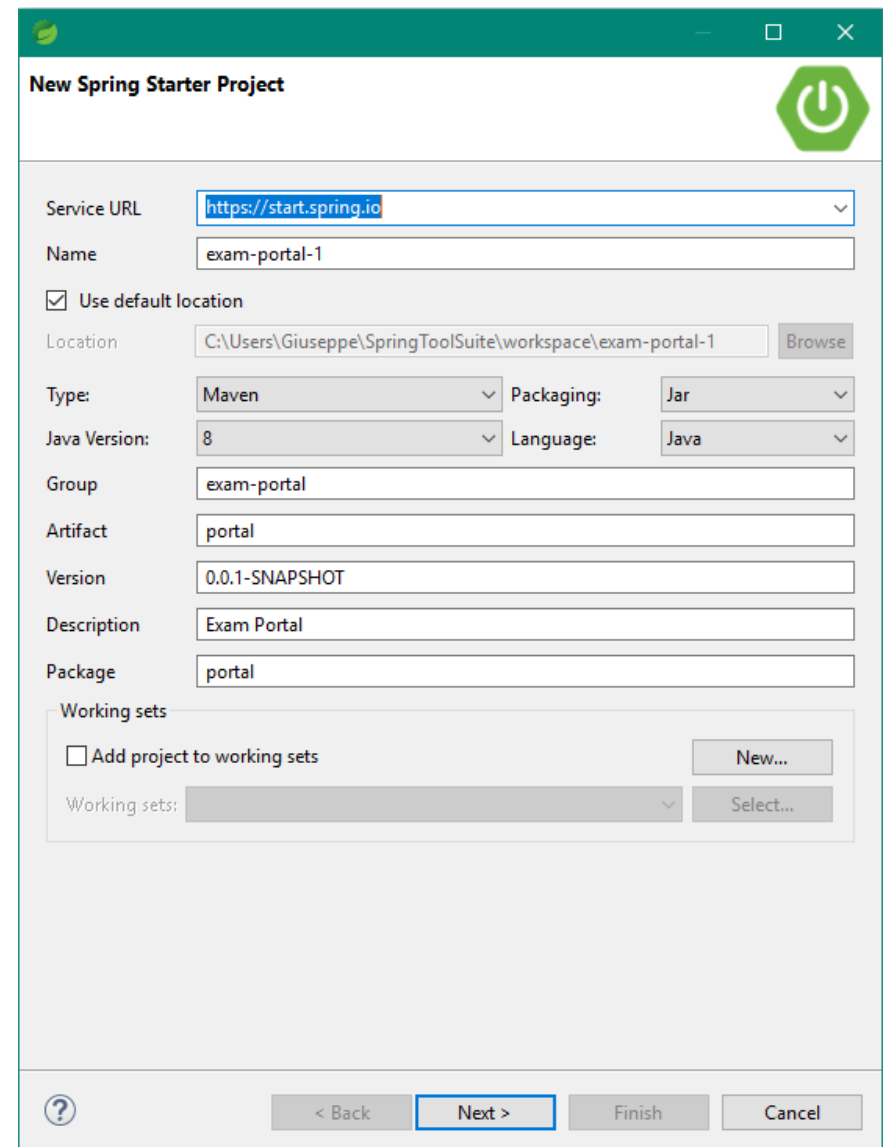
- Spring Tool Suite è una installazione di Eclipse su misura per lo sviluppo Spring
- Integra i componenti necessari o utili per Spring
- Con STS si può creare un progetto Spring Boot direttamente dall'IDE
- Con la creazione guidata, il progetto viene generato su Initializr, scaricato e importato direttamente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



# Creare un progetto usando STS (cont)

Vengono richiesti i dati del progetto da creare tra cui

- nome
- Descrizione
- se usare Maven o Gradle per la gestione delle dipendenze



The screenshot shows the 'New Spring Starter Project' dialog box in Spring Tool Suite. The dialog is titled 'New Spring Starter Project' and features a green power button icon in the top right corner. The fields are as follows:

- Service URL:** A dropdown menu with 'https://start.spring.io' selected.
- Name:** A text field containing 'exam-portal-1'.
- Use default location:** A checked checkbox.
- Location:** A text field showing 'C:\Users\Giuseppe\SpringToolSuite\workspace\exam-portal-1' with a 'Browse' button to its right.
- Type:** A dropdown menu with 'Maven' selected.
- Packaging:** A dropdown menu with 'Jar' selected.
- Java Version:** A dropdown menu with '8' selected.
- Language:** A dropdown menu with 'Java' selected.
- Group:** A text field containing 'exam-portal'.
- Artifact:** A text field containing 'portal'.
- Version:** A text field containing '0.0.1-SNAPSHOT'.
- Description:** A text field containing 'Exam Portal'.
- Package:** A text field containing 'portal'.
- Working sets:** A section with an unchecked checkbox 'Add project to working sets' and a 'New...' button. Below it is a 'Working sets:' dropdown menu and a 'Select...' button.

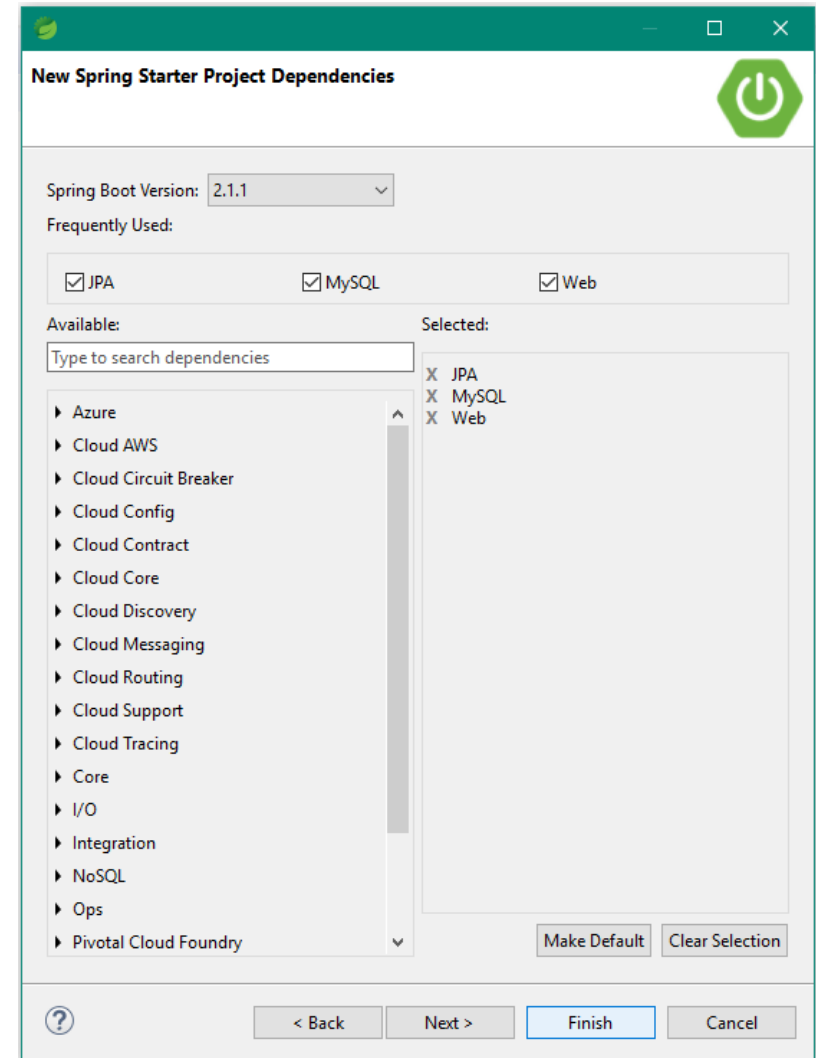
At the bottom of the dialog, there is a question mark icon, a '< Back' button, a 'Next >' button (which is highlighted with a blue border), a 'Finish' button, and a 'Cancel' button.



# Creare un progetto usando STS (cont)

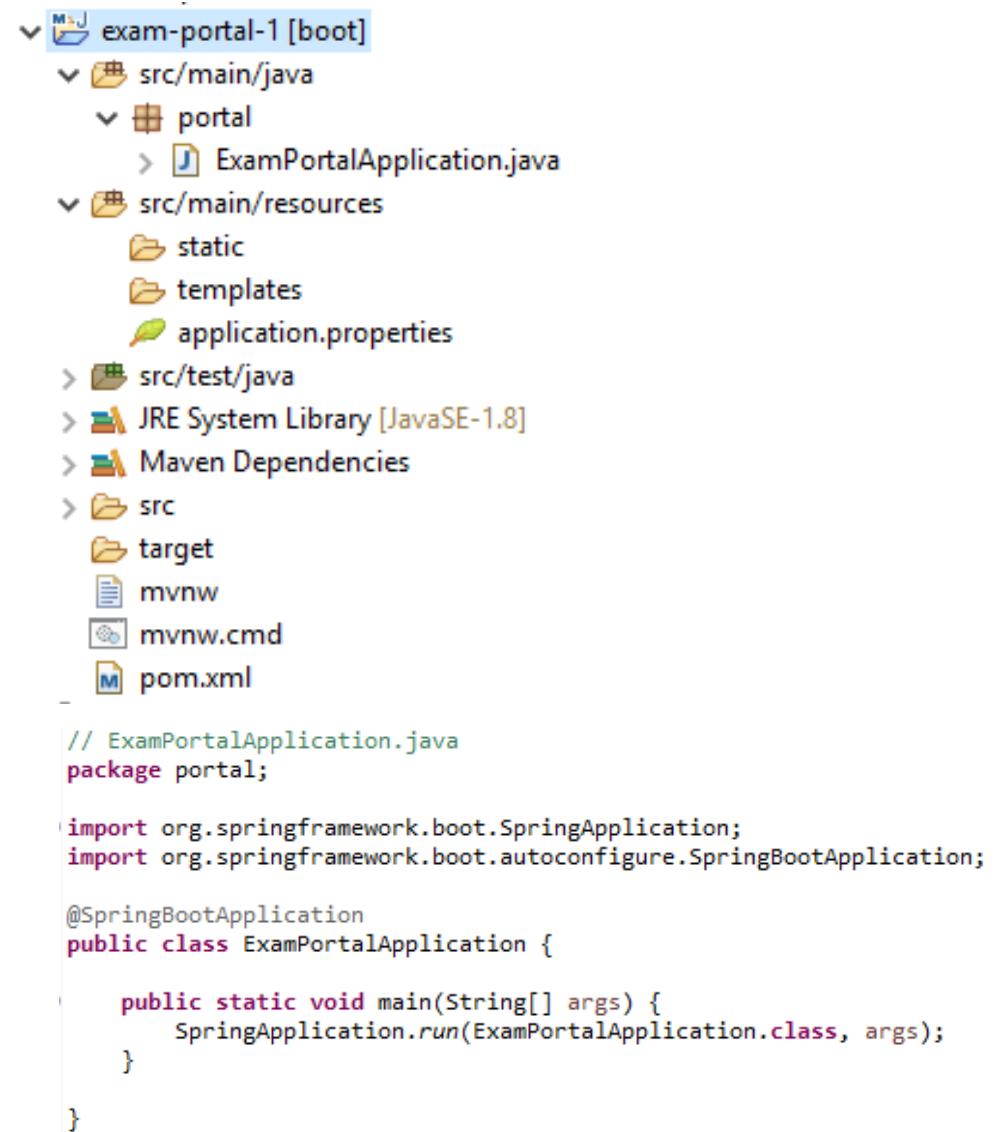
Occorre ancora indicare

- 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
- N.B.: le dipendenze si possono inserire in seguito



# Struttura del progetto creato

- *ExamPortalApplication.java* è la classe che contiene il metodo *main*: inizializza e avvia l'applicazione.
- Il file *pom.xml* (di Maven) dichiara tutte le dipendenze del progetto, che verranno scaricate e aggiunte al progetto automaticamente.
- *application.properties* è il file (inizialmente vuoto) dove vanno inserite tutte le configurazioni aggiuntive che Spring Boot non può configurare automaticamente (o per sovrascrivere le configurazioni di default che usa)



# 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@ ~ $ mysql -u root
Welcome to the MariaDB monitor.  Commands end with ; or \g. Your MariaDB connection id is 195
Server version: 10.3.12-MariaDB HomebrewCopyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

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

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

MariaDB [(none)]> GRANT ALL ON portal.* TO 'portal'@'localhost';
Query OK, 0 rows affected (0.005 sec)
// attenzione a usare host % perche' su OSX sembra che localhost sia gestito diversamente, vedere Linux
```

# Configurazione

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

N.B.: *spring.jpa.hibernate.ddl-auto* può avere i seguenti 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)

```
# application.properties
```

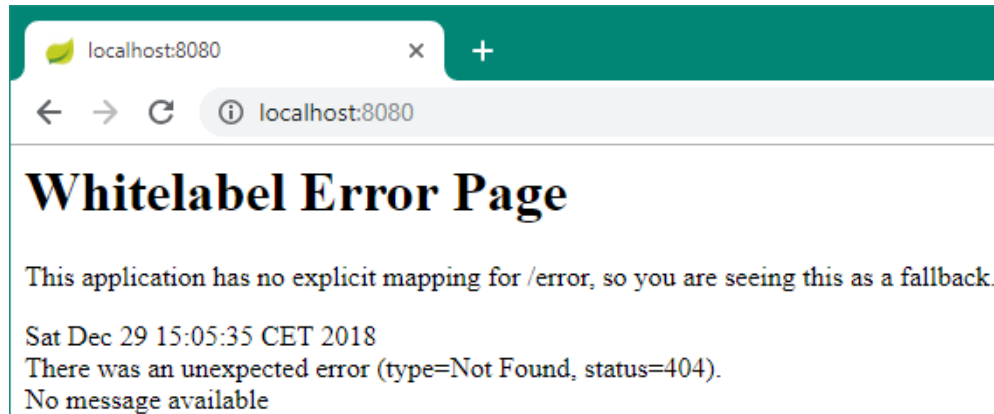
```
spring.jpa.hibernate.ddl-auto=update  
spring.datasource.url=jdbc:mysql://localhost:3306/portal?serverTimezone=Europe/Rome  
spring.datasource.username=portal  
spring.datasource.password=portal
```

# Primo avvio (run)

```
2018-12-29 15:04:33.923 INFO 212 --- [main] portal.ExamPortalApplication : Starting ExamPortalApplication on DESKTOP-C900C5M 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
```

- NB: la password è fonte di problemi; meglio operare con root senza password
  - nel caso, ripartire con *mysqld –initialize-insecure*
- Problemi se un utente appare sia per *localhost* che per l'host % (% in mysql è un *wildcard* degli host)
  - cf. <https://stackoverflow.com/questions/11634084>

# 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* (la proprietà da configurare è *server.port*).

# 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...

NB: sarebbe utile un wizard che generi i template per classi entità come questa

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

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;
    }

    // Getter, Setter e toString
}
```



# Entità Esame (cont)

Riavviando (STOP+RUN) l'applicazione, possiamo notare la tabella associata appena creata e 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)
```

# 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);           1

    Optional<T> findById(ID primaryKey);     2

    Iterable<T> findAll();                   3

    long count();                             4

    void delete(T entity);                   5

    boolean existsById(ID primaryKey);       6

    // ... 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);
    }
}
```

# 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.

# 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);
    }
}
```

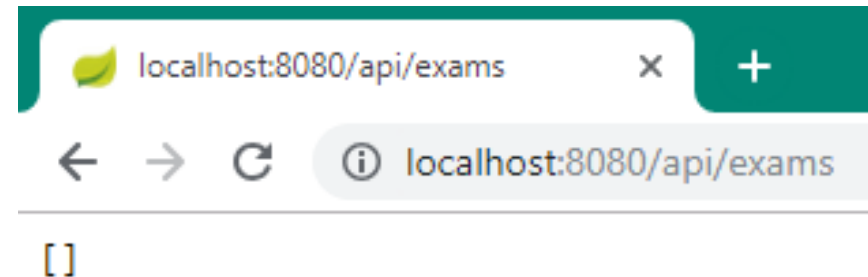


# La richiesta GET per tutti gli esami

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.



# 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.

The screenshot shows the Postman interface for a POST request. The URL is `localhost:8080/api/exams`. The 'Headers' tab is active, showing a table with one header: 'Content-Type' with the value 'application/json'. The 'Body' tab is also visible, showing the 'raw' radio button selected and the JSON body: `{ "description": "Analisi 1", "date": "2019-02-01T15:00:00+0100" }`.

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Content-Type	application/json	
Key	Value	Description

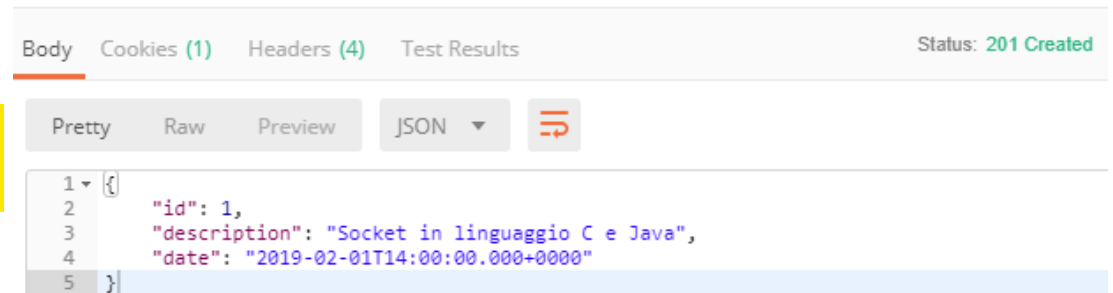
Params Authorization Headers (1) Body Pre-request Script Tests Cookies Code

none form-data x-www-form-urlencoded raw binary JSON (application/json)

```
{ "description": "Analisi 1",  
  "date": "2019-02-01T15:00:00+0100"  
}
```

# 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.

```
mysql> SELECT * FROM exam;
+----+-----+-----+
| id | date           | description                |
+----+-----+-----+
| 1  | 2019-02-01 15:00:00 | Socket in linguaggio C e Java |
+----+-----+-----+
1 row in set (0.00 sec)
```

# 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

GET localhost:8080/api/exams/2

Body Cookies (1) Headers (3) Test Results

Pretty Raw Preview JSON

```
1 {
2   "id": 2,
3   "description": "Thread",
4   "date": "2019-02-04T09:00:00.000+0000"
5 }
```

GET localhost:8080/api/exams/5

Body Cookies (1) Headers (2) Test Results

Status: 404 Not Found

GET localhost:8080/api/exams

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 ]
```

# 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 image shows a sequence of three screenshots from a REST client, illustrating a PUT request and its response. Blue arrows indicate the flow from the first screenshot to the second, and from the second to the third.

**First Screenshot (Top):** A GET request to `localhost:8080/api/exams/2`. The response body is shown in JSON format:

```
{
  "id": 2,
  "description": "Thread",
  "date": "2019-02-04T09:00:00.000+0000"
}
```

**Second Screenshot (Middle):** A PUT request to `localhost:8080/api/exams`. The request body is shown in JSON format:

```
{
  "id": "2",
  "description": "Thread e RMI in Java",
  "date": "2019-02-04T09:30:00+0100"
}
```

**Third Screenshot (Bottom):** A GET request to `localhost:8080/api/exams/2`. The response body is shown in JSON format:

```
{
  "id": 2,
  "description": "Thread e RMI in Java",
  "date": "2019-02-04T08:30:00.000+0000"
}
```



# 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();
}
```

# Esempio di cancellazione

The diagram illustrates a sequence of three REST client screenshots connected by blue arrows, demonstrating a deletion operation.

**First Screenshot (GET Request):**

- Method: GET
- URL: localhost:8080/api/exams/
- Body: Cookies (1), Headers (3), Test Results
- Format: JSON
- Response (Pretty):

```
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 ]
```

**Second Screenshot (DELETE Request):**

- Method: DELETE
- URL: localhost:8080/api/exams/3
- Body: Params, Authorization, Headers, Body, Pre-request Script, Tests
- Status: 200 OK

**Third Screenshot (GET Request):**

- Method: GET
- URL: localhost:8080/api/exams/
- Body: Cookies (1), Headers (3), Test Results
- Format: JSON
- Response (Pretty):

```
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 ]
```

# Struttura finale



# 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.

# Entità Result

A differenza dell'entità Esame, qui serve qualche modifica in più. Abbiamo un attributo Exam annotato con @ManyToOne, questa annotazione indica la relazione tra Result ed Exam da parte di Result (si possono utilizzare anche le annotazioni @OneToMany, @OneToOne per definire le relazioni). L'attributo exam verrà tradotto nell'id dell'esame associato al risultato.

Sarà il supporto JPA a introdurre nel DB mysql le relazioni appropriate che derivano dalle annotazioni riguardanti le relazioni.

Come detto precedentemente, dobbiamo aggiungere il vincolo sulla coppia <studente, esame> poiché deve essere unica all'interno della tabella. Per fare ciò usiamo l'annotazione @Table: con il parametro name indichiamo il nome della tabella; con uniqueConstraints e la relativa annotazione indichiamo che la coppia «student, exam\_id» deve essere unica.

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

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

import javax.persistence.Entity;

@Entity
@Table(name = "result", uniqueConstraints = { @UniqueConstraint(columnNames = { "student", "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() {
    }

    public Result(Exam exam, String student, String mark, String note) {
        this.exam = exam;
        this.student = student;
        this.mark = mark;
        this.note = note;
    }

    // Getter, Setter e toString
}
```

# 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();
}
```

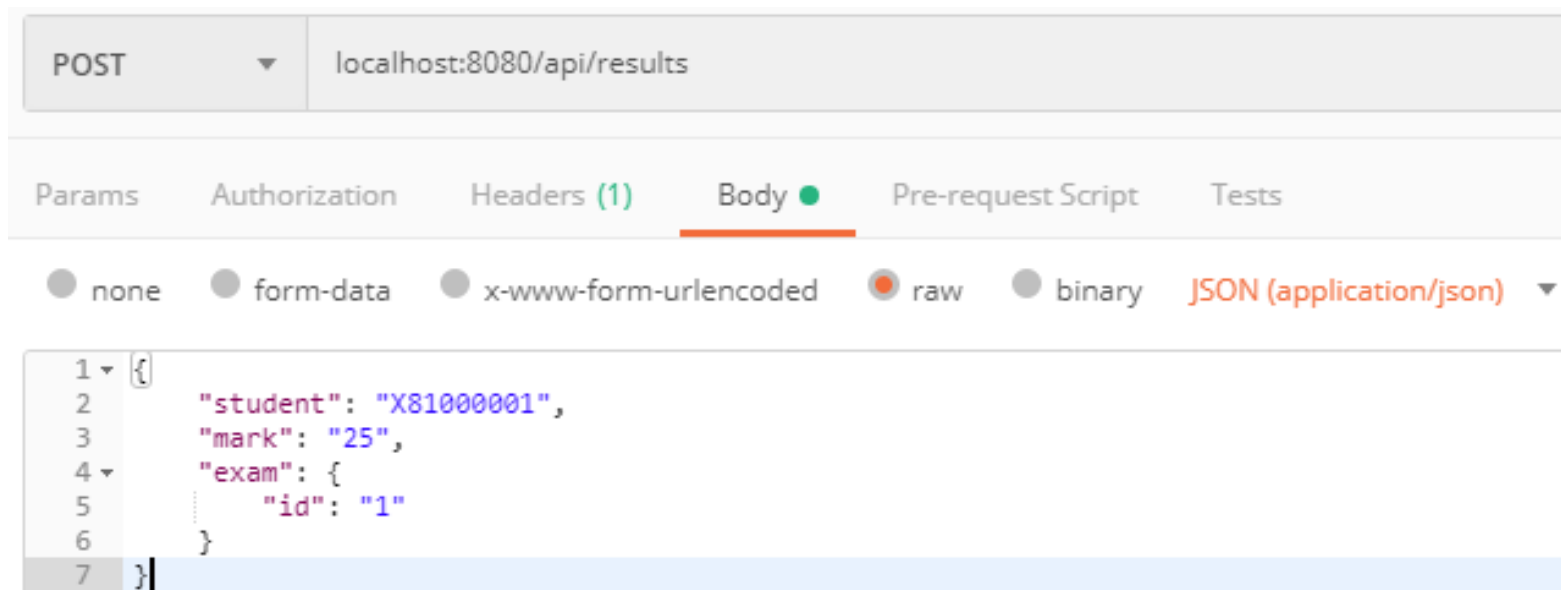


# Aggiungere un risultato

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



# Tutti i risultati di tutti gli esami

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

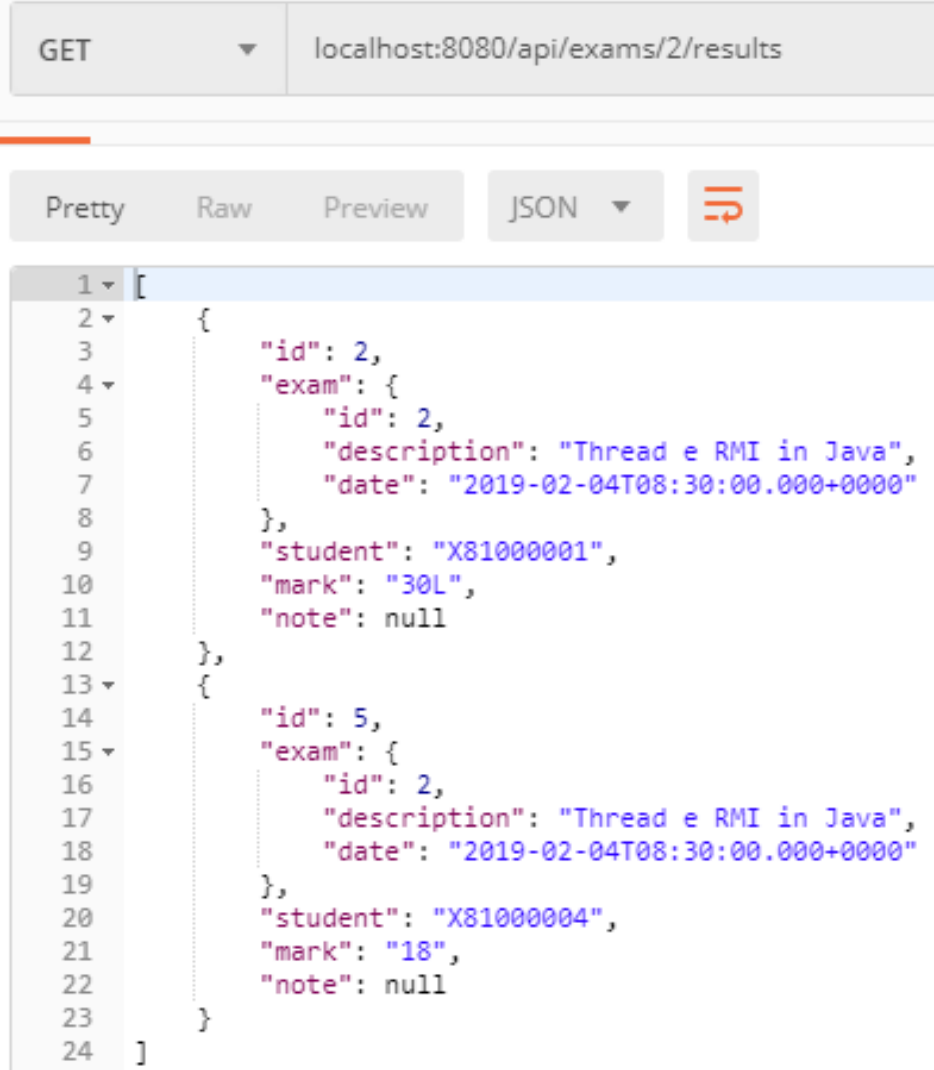


The screenshot shows a web browser interface with a GET request to `localhost:8080/api/results`. The response is displayed in JSON format, showing a list of exam results. The JSON structure is as follows:

```
[{"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}]
```

# Tutti i risultati di un esame

Esempio: tutti i risultati dell'esame con id 2



The screenshot shows a REST client interface with a GET request to `localhost:8080/api/exams/2/results`. The response is displayed in JSON format, showing two exam results for exam ID 2. The first result is for student `X81000001` with a mark of `30L`. The second result is for student `X81000004` with a mark of `18`. Both results are for exam ID 2, which has the description "Thread e RMI in Java" and the date "2019-02-04T08:30:00.000+0000".

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

Pretty Raw Preview JSON

[
  {
    "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
  }
]
```

# Tutti i risultati di uno studente

Esempio: tutti i risultati dello studente con matricola X81000001



The screenshot shows a REST client interface with a GET request to `localhost:8080/api/user/X81000001/results`. The response is displayed in JSON format, showing two exam results for the student with 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": "30L",
    "note": null
  }
]
```