



Spring

H2





Database in-memory H2

- I database in-memory sono delle sorgenti dati che vengono spesso utilizzate per la fase di testing.
- Le tabelle di un DB in-memory vengono cancellate quando l'applicazione si arresta(comportamento di default).
- Spring Boot fornisce un supporto completo ad H2.
- H2 è configurabile tramite proprietà da applicare nell'application.properties.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```



Application properties



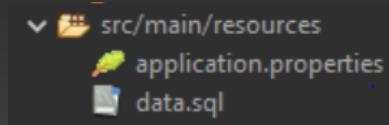
```
spring.datasource.url=jdbc:h2:mem:dbprova
spring.datasource.dbcp2.driver-class-name=org.h2.Driver
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.show-sql=true
spring.jpa.database-
platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
#sarà il path utilizzato per la console del database
spring.h2.console.path=/consoleDB
#tramite quest'istruzione comunichiamo al db di eseguire
le istruzioni
#presenti nel data.sql dopo la creazione del db
spring.jpa.defer-datasource-initialization=true
```





Inizializzazione del database

- Creare un file «.sql» nella cartella «src/main/resources».



```
1 INSERT INTO RUOLO (nome) VALUES ('admin');
2 INSERT INTO RUOLO (nome) VALUES ('user');
3
4 INSERT INTO UTENTE (email,eta,nome) VALUES('matiasasis@gmail.com', 23, 'matias');
5 INSERT INTO UTENTE (email,eta,nome) VALUES('saraverdi@gmail.com', 27, 'sara');
6
7 INSERT INTO UTENTE_RUOLI(UTENTI_ID, RUOLI_ID) VALUES(1, 1);
8 INSERT INTO UTENTE_RUOLI(UTENTI_ID, RUOLI_ID) VALUES(2, 2);
```





Database in-memory H2

- I database in-memory sono delle sorgenti dati che vengono spesso utilizzate per la fase di testing.
- Le tabelle di un DB in-memory vengono cancellate quando l'applicazione si arresta(comportamento di default).
- Spring Boot fornisce un supporto completo ad H2.
- H2 è configurabile tramite proprietà da applicare nell'application.properties.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```





Database in-memory H2

- I database in-memory sono delle sorgenti dati che vengono spesso utilizzate per la fase di testing.
- Le tabelle di un DB in-memory vengono cancellate quando l'applicazione si arresta(comportamento di default).
- Spring Boot fornisce un supporto completo ad H2.
- H2 è configurabile tramite proprietà da applicare nell'application.properties.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```



Console H2



- La console per la gestione del DB è raggiungibile di default all'URL:
`http://localhost:8080/h2-console`
- Ma tramite questa configurazione:
 - E' possibile raggiungerla all'URL: `http://localhost:8080/consoleDB`

```
spring.h2.console.path=/consoleDB
```

A screenshot of a web browser displaying the H2 Console login interface. The URL in the address bar is `localhost:8080/consoleDB/test.do?sessionid=10a67d1bd67574c22d660ff0981cefb8`. The page has a blue header bar with dropdown menus for "English", "Preferences", "Tools", and "Help". Below this is a "Login" form. The "Saved Settings" dropdown is set to "Generic H2 (Embedded)". The "Setting Name" field contains "Generic H2 (Embedded)" with "Save" and "Remove" buttons next to it. The "Driver Class" field is set to "org.h2.Driver". The "JDBC URL" field contains "jdbc:h2:mem:dbprova". The "User Name" field is set to "test". The "Password" field is empty. At the bottom of the form are "Connect" and "Test Connection" buttons. A green footer bar at the bottom of the page displays the message "Test successful".

English Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded)

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:dbprova

User Name: test

Password:

Connect Test Connection

Test successful



Console H2



Auto commit Max rows: 1000 | Auto complete Off | Auto select On | ?

Run Run Selected Auto complete Clear SQL statement:

jdbc:h2:mem:dbprova

- RUOLO
- UTENTE
- UTENTE_RUOLI
- INFORMATION_SCHEMA
- Users

H2 2.1.214 (2022-06-13)

Important Commands

	Displays this Help Page
	Shows the Command History
	Ctrl+Enter Executes the current SQL statement
	Shift+Enter Executes the SQL statement defined by the text selection
	Ctrl+Space Auto complete
	Disconnects from the database

Sample SQL Script

Delete the table if exists	DROP TABLE IF EXISTS TEST;
Create a new table with ID and NAME columns	CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));
Add a new row	INSERT INTO TEST VALUES(1, 'Hello');
Add another row	INSERT INTO TEST VALUES(2, 'World');
Query the table	SELECT * FROM TEST ORDER BY ID;
Change data in a row	UPDATE TEST SET NAME='Hi' WHERE ID=1;
Remove a row	DELETE FROM TEST WHERE ID=2;
Help	HELP ...

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM UTENTE

SELECT * FROM UTENTE;

ID	EMAIL	ETA	NOME
1	matiasasis@gmail.com	23	matias
2	saraverdi@gmail.com	27	sara

(2 rows, 1 ms)

Edit





Spring

Pagination E Sorting





Sorting

- Tutti i metodi di JpaRepository possono prendere un Ingresso un parametro di tipo Sort, possiamo crearlo tramite il metodo Sort.By("nomeAttributoJava") chiamando poi o il .ascending() se vogliamo un ordine crescente o il .descending() se vogliamo un ordine decrescente

```
public Sort getSortAscending(String primoParametro, String ...altriParametri){  
    Sort s=Sort.by(primoParametro).ascending();  
    if(altriParametri==null) return s;  
    for(String s1:altriParametri){  
        s=s.and(Sort.by(s1).ascending());  
    }  
    return s;  
}
```



Pagination

- Tutti i metodi di JpaRepository possono prendere un Ingresso un parametro di tipo Pageable, possiamo crearlo tramite il metodo statico of della classe PageRequest, ne esistono due implementazioni, una che prende il numero di pagina e quanti elementi inserire, nella seconda abbiamo anche un Sorting

```
Pageable p1=PageRequest.of(numeroPagina,numeroElementi);
Pageable p2=PageRequest.of(numeroPagina,numeroElementi, s);
```





Spring

Optimistic Locking



Concorrenza nei Sistemi di Gestione dei Dati



- Quando più utenti cercano di accedere e modificare gli stessi dati contemporaneamente.
- Questo può generare dei conflitti e inconsistenza dei dati.
- La soluzione è l'Optimistic Locking, ovvero una tecnica per evitare conflitti e garantire l'integrità dei dati.



Concetto di Versione (Versioning)

- L'Optimistic Locking utilizza una "versione" associata ai dati.
- La versione è un valore numerico che cambia ogni volta che i dati vengono modificati.
- Prima di salvare una modifica, Spring verifica se la versione dei dati è la stessa di quando sono stati letti.





Implementazione in Spring

- In Spring, è possibile utilizzare l'annotazione `@Version` per gestire la versione dei dati.
 - `@Version private long version;`
- Questa annotation permette di abilitare l' Optimistic Locking tramite JPA e Hibernate.



Gestione delle Eccezioni



- Se due utenti cercano di modificare gli stessi dati contemporaneamente, si verifica un conflitto.
- Spring lancia un'eccezione `ObjectOptimisticLockingFailureException` in caso di conflitto.
- Gestiremo questa eccezione in modo appropriato.



Esempio - Implementazione in Spring con JPA



```
@Entity  
public class Prodotto {  
    @Id  
    private long id;  
    private String nome;  
    private double prezzo;  
    @Version  
    private long version;  
}
```



Esempio - Implementazione in Spring con JPA



```
try {
    ProdottoService service=new ProdottoService();
    Prodotto p = service.findById(1); // Legge un prodotto
    p.setPrezzo(29.99); // Modifica il prezzo
    service.salva(p); // Tenta di salvare il prodotto
} catch (ObjectOptimisticLockingFailureException ex) {
    // Gestione del conflitto
    throw new ResponseStatusException(HttpStatus.CONFLICT, "qualcun altro ha modificato il prodotto");
}
```



Vantaggi dell'Optimistic Locking



- Migliora la concorrenza: Consente a più utenti di accedere ai dati contemporaneamente senza bloccare l'accesso.
- Minimizza i conflitti: Riduce la probabilità di conflitti e rallentamenti nel sistema.
- Maggiore Scalabilità: Favorisce una migliore scalabilità delle applicazioni.





Spring

Pessimistic Locking





Concetto

- Il pessimistic locking utilizza meccanismi del database per riservare accesso esclusivo ai dati.
- Questo approccio assume che la contesa per un elemento di dati accadrà e previene i conflitti bloccando l'elemento per tutta la durata della transazione.
- Una volta che un thread ha acquisito un lock, nessun altro thread può accedere ai dati fino al rilascio del lock. Tutti i lock sono mantenuti fino al commit o rollback della transazione.





Lock Modes

LOCK MODE	TIPO LOCK	COMPORTAMENTO	USO PRINCIPALE
PESSIMISTIC_READ	Shared	Previene update e delete dei dati, ma permette la lettura. Se il database non supporta questo mode, viene automaticamente usato PESSIMISTIC_WRITE.	Lettura senza dirty reads quando non servono modifiche immediate
PESSIMISTIC_WRITE	Exclusive	Previene lettura, update e delete secondo la specifica JPA. Alcuni database con multi-version concurrency control permettono comunque ai lettori di recuperare dati bloccati.	Modifiche con massimo isolamento in scenari ad alta contesa
PESSIMISTIC_FORCE_INCREMENT	Exclusive + Version	Funziona come PESSIMISTIC_WRITE ma incrementa anche l'attributo @Version delle entità versionate. Acquisire questo lock risulta nell'aggiornamento della colonna version.	Cooperazione con entità versionate, combinando strategie pessimistiche e ottimistiche





Implementazione

- @Lock può essere applicata a metodi di query personalizzati e metodi CRUD
- Per applicare lock ai metodi CRUD predefiniti, ridichiarare il metodo nell'interfaccia
- @Transactional è essenziale: il lock viene mantenuto per tutta la durata della transazione
- Il lock viene rilasciato automaticamente al commit o rollback



Esempio - Implementazione JpaRepository



```
public interface ProdottoRepository extends JpaRepository<Prodotto, Long> {  
    @Lock(LockModeType.PESSIMISTIC_WRITE)  
    Optional<Prodotto> findById (long id);  
    @Lock(LockModeType.PESSIMISTIC_READ)  
    List<Prodotto> findByCategory(String category);  
}
```



Esempio - Implementazione nel Service



```
@Transactional  
public void aggiornaPrezzo(long id, double nuovoPrezzo) {  
    Prodotto p = repo.findByIdLocked(id)  
        .orElseThrow(()->new  
ResponseStatusException(HttpStatus.CONFLICT));  
    p.setPrezzo(nuovoPrezzo);  
}
```





Spring

Junit 5



JUnit 5 – Cos'è



- JUnit è un framework open source, usato per scrivere ed eseguire unit testing per Java.
- Attualmente siamo alla versione 5 che differisce dalle versioni precedenti per la sua composizione
- La versione 5 di JUnit è composta da 3 macro progetti
 - JUnit Platform
 - JUnit Jupiter
 - JUnit Vintage



JUnit 5 – JUnit Platform



- JUnit Platform funge da base per l'avvio di framework di test sulla JVM.
- Definisce inoltre l'API TestEngine per lo sviluppo di un framework di test che viene eseguito sulla piattaforma.
- Inoltre, la piattaforma fornisce un Console Launcher per avviare la piattaforma dalla riga di comando e JUnit Platform Suite Engine per eseguire una suite di test personalizzata utilizzando uno o più motori di test sulla piattaforma.
- Negli IDE più diffusi è già integrato un supporto per JUnit(vedi IntelliJ IDEA, Eclipse, NetBeans e Visual Studio Code) così come negli strumenti di compilazione (vedi Gradle, Maven e Ant).



JUnit 5 – JUnit Vintage



- JUnit Vintage fornisce un TestEngine per l'esecuzione di test basati su JUnit 3 e JUnit 4.
- Per utilizzarlo è indispensabile che JUnit 4.12 o una versione successiva sia presente nel module path.



JUnit 5 – JUnit Jupiter



- JUnit Jupiter è il «nuovo» sistema di scrittura per i test in JUnit
- Fornisce l'integrazione tra i test scritti in JUnit 5 e il codice esistente

JUnit 5 – Configurazione



- Per iniziare a testare le nostre applicazioni con JUnit abbiamo bisogno di uno starter
- Lo starter che ci permetterà di scrivere il nostro codice sarà lo Starter-Test

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

- Attenzione: in questo starter definiremo anche lo scope, in modo da segnalare al progetto che il package di riferimento non è il main ma il test





JUnit 5 – Configurazione

- Oltre allo starter di SpringBoot se vogliamo testare la nostra security dovremo importare anche

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

- Attenzione: anche in questo caso definiremo lo scope, in modo da segnalare al progetto che il package di riferimento non è il main ma il test



JUnit 5 – Classe di Partenza



- Come per un progetto Spring standard anche per quanto riguarda JUnit avremo nel nostro progetto una classe di partenza
- Questa classe avrà tre annotation ben distinte che la contraddistingueranno
- `@SpringBootTest`
- `@ContextConfiguration`
- `@TestMethodOrder`

JUnit 5 – SpringBootTest



- L'annotation `@SpringBootTest` è l'annotazione principale che segnala lo start del nostro test



JUnit 5 – @ContextConfiguration



- L'annotation `@ContextConfiguration` definisce la classe di `ApplicationContext` del nostro progetto Spring per i test di integrazione.
 - Prima di Spring 3.1, erano supportate solo le risorse basate su path di file xml.
 - Da Spring 3.1, possiamo scegliere di basare i nostri test o su path o su classi, non entrambe
 - Da Spring 4.0.4, possiamo basare i nostri test su entrambe le cose decidendo di volta in volta cosa vogliamo testare
 - Quindi ad oggi possiamo inserire sia le classi con l'attributo `classes`, sia i path con l'attributo `location`.
 - Si preferisce tuttavia basare i test su classi in modo da utilizzare solo un singolo tipo di risorsa.
 - Le posizioni delle risorse basate sul percorso possono essere file di configurazione XML ma, i framework di terze parti possono scegliere di supportare ulteriori tipi di risorse basate sul percorso.



JUnit 5 – @TestMethodOrder



- L'annotation `@TestMethodOrder` definisce l'ordine di esecuzione dei nostri test
- Come parametro gli settiamo `OrderAnnotation` di `org.junit.jupiter.api.MethodOrderer.OrderAnnotation`;
- Si riferirà alle annotation interne che contraddistingueranno i singoli metodi
- Nel caso in cui due metodi hanno la stessa precedenza verranno eseguiti in maniera casuale
- I metodi ai quali non è stato assegnato un valore esplicito verranno eseguiti dopo dei metodi ai quali è stato assegnato un valore esplicito



JUnit 5 – TestApplication



```
@SpringBootTest  
@ContextConfiguration( classes = PrimoProgettoApplication.class)  
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)  
class PrimoProgettoApplicationTests {  
    //Metodi di test  
}
```



JUnit 5 – MockMvc



- Per testare i nostri servizi avremo bisogno di un oggetto del tipo MockMvc
- questa classe è l'Entry Point per il supporto server-side del testing
- Per crearla oltre all'autowired andremo ad aggiungere sulla classe l'annotation @AutoConfigureMockMvc



JUnit 5 – TestApplication



```
@SpringBootTest  
@ContextConfiguration( classes = PrimoProgettoApplication.class)  
@TestMethodOrder(OrderAnnotation.class)  
@AutoConfigureMockMvc  
class PrimoProgettoApplicationTests {  
    //Metodi di test  
}
```



JUnit 5 – scrittura del test



- Una volta finita la configurazione possiamo creare il nostro metodo di test sul quale andremo a mettere l'annotation @Test
- Altre annotation opzionali possono essere l'ordine di esecuzione definito da @Order e la gestione della security
- All'interno del metodo utilizzeremo il metodo perform dell'oggetto di tipo mock che preso in ingresso un RequestBuilder ci permetterà di ottenere una ResultAction
- Ovvero una classe che tramite il metodo andExpect ci permetterà di segnalare al nostro test il risultato aspettato

JUnit 5 – metodo di test



```
@Order(1)
@Test
public void TestLoginSenzaUtente() throws Exception{
    mock.perform(MockMvcRequestBuilders.post("/all/login")
                .accept(MediaType.APPLICATION_JSON))
        .andExpect(MockMvcResultMatchers.status().isBadRequest())
        .andReturn();

}
```



JUnit 5 – sicurezza nel test



- Se sto testando anche la sicurezza le possibilità per i metodi che implementano la security JWT sono fondamentalmente due
- La prima, più macchinosa è quella di effettuare la login tramite un mock prima della chiamata e passare l'authentication token alla chiamata che ci interessa (funziona anche senza importare spring security test)
- La seconda è quella di annotare i nostri metodi con `@WithUserDetails("usernameDellUtente")`, questo ci permetterà di "automatizzare" la login
- Per i metodi senza security basterà invocarli



JUnit 5 – test dei metodi



```
@Order(1)
@Test
public void TestLoginSenzaUtente() throws Exception{
    mock.perform(MockMvcRequestBuilders.post("/all/login")
                .accept(MediaType.APPLICATION_JSON))
        .andExpect(MockMvcResultMatchers.status().isBadRequest())
        .andReturn();

}
```





JUnit 5 – creazione del MockMvc

```
@Order(1)
@Test
@WithUserDetails("Antonio523")
public void testAccessoConDiritti() throws Exception{
    mock.perform(MockMvcRequestBuilders.get("/admin/visualizzaRichieste")
                .accept(MediaType.APPLICATION_JSON))
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andReturn();

}
```





JUnit 5 – creazione del MockMvc

```
@Order(1)
@Test
public void provaSecurityConToken() throws Exception {
    LoginRequest request=new LoginRequest();
    request.setEmail("Antonio523");
    request.setPassword("Password!1");
    String json=new ObjectMapper().writeValueAsString(request);
    String tokenJwt=mock.perform(MockMvcRequestBuilders.post("/all/login")
        .contentType(MediaType.APPLICATION_JSON_VALUE)
        .content(json))
        .andReturn().getResponse().getHeader("Authorization");
    mock.perform(MockMvcRequestBuilders.get("/admin/ visualizzaRichieste ")
        .header("Authorization","Bearer "+tokenJwt)
    )
    .andExpect(MockMvcResultMatchers.status().is(200))
    .andReturn();
}
```





JUnit 5 – parametri nel body

- Per passare dei parametri nel body dovremo passare sia il contentType che il content al RequestBuilder
- Questo passaggio lo faremo convertendo i nostri oggetti tramite la libreria autoimportata da spring Jackson tramite l'ObjectMapper

```
@Order(1)
@Test
public void testLoginConUtente() throws Exception{
    String json=new ObjectMapper().writeValueAsString(new
LoginRequest("admin@email.com","root"));
    mock.perform(MockMvcRequestBuilders.post("/all/login")
        .contentType(MediaType.APPLICATION_JSON)
        .content(json)
        .accept(MediaType.APPLICATION_JSON))
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andReturn();
}
```



JUnit 5 – distribuzione dei metodi



- Consideriamo sempre per una corretta distribuzione dei test che possiamo creare quante classi vogliamo annotate con @SpringBootTest quindi creeremo una classe di test per ogni Controller che abbiamo



JUnit 5 – controllare il risultato



- Un'altra possibilità che abbiamo con JUnit è quella di controllare il risultato di un endpoint
- Per farlo useremo la classe MockMvcResultMatchers che tramite il metodo jsonPath ci permetterà di controllare i singoli valori del Json risultante



JUnit 5 – controllare il risultato



```
@Test
@Order(1)
@WithUserDetails("Antonio523")
public void controllaJsonDiRisultato() throws Exception
{
    mock.perform(MockMvcRequestBuilders.get("/admin/visualizzaValore")
                .accept(MediaType.APPLICATION_JSON))
        .andExpect(MockMvcResultMatchers.status().isOk())

    .andExpect(MockMvcResultMatchers.content().contentTypeCompatibleWith(MediaType
    .APPLICATION_JSON))
        .andExpect(MockMvcResultMatchers.jsonPath("$.nomeCampo").exists())
    .andExpect(MockMvcResultMatchers.jsonPath("$.nomeCampo").value("valore"))
    .andExpect(MockMvcResultMatchers.jsonPath("$.nomeAttributo.nomeCampo").exists(
))
    .andExpect(MockMvcResultMatchers.jsonPath("$.nomeAttributo.nomeCampo").value(
"valore"))
        .andDo(MockMvcResultHandlers.print());
}
```

JUnit 5 – inserimento



```
@Test  
 @Order(1)  
 public void testInsArticolo() throws Exception  
{  
  
 mock.perform(MockMvcRequestBuilders.post("/api/articoli/inserisci")  
 .contentType(MediaType.APPLICATION_JSON)  
 .content(JsonData)  
 .accept(MediaType.APPLICATION_JSON))  
 .andExpect(MockMvcResultMatchers.status().isCreated())  
  
 .andExpect(jsonPath("$.data").value(LocalDate.now().toString()))  
 .andExpect(jsonPath("$.message").value("Inserimento  
 Articolo Eseguita con successo!"))  
 .andDo(print());  
}
```



JUnit 5 – controllare i service



- Un'altra possibilità che abbiamo con JUnit è quella di controllare direttamente i metodi dei service
- Per farlo avendo bisogno sia dei service che potremo importare come `@Autowired`
- Sia dei metodi statici di due classi, entrambe chiamate Assertions faremo l'import statico di tutti i metodi che ci interessano
- Le classi da cui fare gli import statici sono
 - `org.assertj.core.api.Assertions`
 - `org.junit.jupiter.api.Assertions`

JUnit 5 – controllare le repository



- Tramite il primo metodo potremo controllare che l'elenco dei valori risultanti abbia la lunghezza che ci aspettiamo

```
@Test  
{@Order(1)  
public void testSulleListe(){  
    List<Utente> items = repo.visulizzaRichieste();  
    assertEquals(5, items.size());  
}}
```



JUnit 5 – controllare le repository



- Tramite il secondo metodo potremo controllare che uno dei valori dell'oggetto risultante della query abbia il valore che ci aspettiamo

```
@Test  
{@Order(4)  
public void TestSingoloAttributo(){  
  
assertThat(repo.findUtenteByEmail("admin@email.com")).get()  
    .extracting(Utente::getUsername)  
    .isEqualTo("admin");  
}}
```

