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

spring boot

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
- @ContexConfiguration
- @TestMethodOrder

JUnit 5 – SpringBootTest

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

JUnit 5 – @ContexConfiguration

- L’annotation @ContextConfiguration definisce la classe di ApplicationContex 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 @AutoCofingureMockMvc

spring boot

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

```java
@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 – 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 – 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)

JUnit 5 – creazione del test

```java
@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 – creazione del test

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

}
```

JUnit 5 – test dell'endpoint GraphQL

- Se sto testando gli endpoint di grahql devo utilizzare due oggetti diversi, uno per i metodi senza autenticazione e uno per quelli con autenticazione
- Entrambi verranno creati tramite un metodo annotato con @BeforeEach che verrà eseguito automaticamente prima di ogni test
- L'intero applicativo sarà eseguito su una porta random libera nel nostro applicativo

spring boot

JUnit 5 – creazione dei GraphQlTester

```java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
    public class TestGraphQL {

        @LocalServerPort
        int port;

        private GraphQlTester testerSenzaAuth,testerConAuth;


        @Autowired
        GestoreTokenService service;

        @BeforeEach
        void setUp() {
            WebTestClient.Builder builder =  WebTestClient
                    .bindToServer()
                    .baseUrl("http://localhost:" + port + "/graphql");

            testerSenzaAuth = HttpGraphQlTester
                    .builder(builder)
                    .build();

            builder =  WebTestClient
                    .bindToServer()
                    .defaultHeader("Authorization", "Bearer "+getToken())
                    .baseUrl("http://localhost:" + port + "/graphql");

            testerConAuth = HttpGraphQlTester
                    .builder(builder)
                    .build();
        }

    }
```

```java
private String getToken(){
        Utente u=new Utente();
        u.setNome("mario");
        u.setCognome("rossi");
        u.setEmail("m.rossi@email.com");
        u.setRuolo(Ruolo.UTENTE);
        return service.generaToken(u);
    }
```

JUnit 5 – test con GraphQL

```
@Test
    void testQuery() {
        String body="""
                query Login {
                    login(input: { username: "m.rossi@email.com", password: "1234" }) {
                        token
                        utente {
                            nomeCompleto
                        }
                    }
                }
            """;
        testerSenzaAuth
                .document(body)
                .execute()
                .path("login.utente.nomeCompleto")
                .entity(String.class)
                .isEqualTo("mario rossi");
    }
```