Università degli Studi di Bari Aldo Moro

A.A. 2024/2025

- L.M. SICUREZZA INFORMATICA -

Sicurezza delle Architetture Orientate ai Servizi

- *Utilizzo del token JWT con Spring Boot* -

Professore: MALLARDI Giulio

Studente: DAMMACCO Cataldo

Data: 08/09/2025

Sommario

1. Spring

Spring è un framework di sviluppo Java open source leggero che fornisce un modello completo di programmazione e configurazione per lo sviluppo di applicazioni Java a livello *enterprise*, semplificando lo sviluppo ed aiutando così gli sviluppatori a creare applicazioni in modo più efficace ed efficiente.

Spring si concentra su diverse aree di sviluppo delle applicazioni e offre un'ampia gamma di funzionalità come *Dependency Injection* e **moduli** pronti all'uso che sono:

- Spring MVC;
- Spring JDBC;
- Spring Web Flow;
- Spring Security;
- Spring ORM;
- Spring AOP;
- Spring Test.

Questi moduli offrono migliori funzionalità per le applicazioni *web* e riducono drasticamente i tempi di sviluppo. Ad esempio, all'inizio dello sviluppo *Web Java*, era necessario scrivere molte linee di codice *standard* per l'inserimento di un record in un database. Utilizzando il *JDBCTemplate* del modulo Spring JDBC, è possibile ridurlo a poche righe di codice con semplici configurazioni.

1.1. Cos'è Spring Boot?

Spring Boot è un'estensione di Spring, che elimina le configurazioni standard richieste per impostare un'applicazione Spring. Dotato di codice predefinito e configurazione basata su annotazioni, Spring Boot consente un ecosistema di sviluppo più rapido ed efficiente.

Poiché Spring Boot è costruito sulla base di Spring, offre tutte le funzionalità e i vantaggi di Spring. Spring Boot mira a ridurre la lunghezza del codice e offre agli sviluppatori il modo più semplice per creare un'applicazione.

1.2. Dipendenze Maven

Prima di tutto, diamo un'occhiata alle dipendenze minime richieste per creare un'applicazione web utilizzando Spring:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>5.3.18</version>
</dependency>
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.3.18</version>
</dependency>
```

A differenza di Spring, Spring Boot richiede solo una dipendenza per far funzionare un'applicazione web:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.6.6</version>
</dependency>
```

1.3.Spring MVC

Spring MVC è un modulo del framework Spring che implementa il pattern **Model-View-Controller** (**MVC**), organizzando il codice in tre parti fondamentali:

- *Model* → rappresenta i dati e la logica di *business* dell'applicazione.
- *View* → rappresenta ciò che l'utente vede, cioè l'interfaccia di output.
- *Controller* → gestisce le richieste dell'utente e coordina Model e View.

In Spring MVC il *DispatcherServlet* funge da controller centrale: riceve ogni richiesta HTTP, decide quale controller deve gestirla e quale risposta restituire.

Nel progetto SAOS2025, sviluppato con Spring Boot, l'approccio è orientato alle API REST:

- *Model* → rappresentato dalle entità JPA (Utente, Libro, Ruolo), dai repository e dalla logica di accesso ai dati.
- *View* → non è basata su JSP ma sulle risposte JSON restituite dai controller, utilizzabili da qualsiasi client (Postman, HTML+JavaScript, applicazioni mobile, ecc.).
- *Controller* → sono classi annotate con @RestController che espongono endpoint HTTP per eseguire operazioni CRUD e gestire l'autenticazione.

Configurazione in Spring Boot

A differenza delle applicazioni Spring MVC classiche (con JSP), in Spring Boot non è necessario configurare manualmente DispatcherServlet o ViewResolver. Spring Boot autoconfigura il supporto MVC, mappa automaticamente i controller e gestisce la serializzazione/deserializzazione JSON tramite Jackson.

Un esempio di controller REST nel progetto:

```
@RestController
@RequestMapping("/api/libro")
public class LibroController {

   @Autowired
   private LibroRepository libroRepository;

   @GetMapping("/findAllBooks")
   @PreAuthorize("hasRole('UTENTE') or hasRole('MODERATORE') or hasRole('ADMIN')")
   public List<Libro> findAllBooks() {
      return libroRepository.findAll();
   }
}
```

In questo caso:

- Non c'è una vista JSP: il metodo findAllBooks() restituisce direttamente una lista di oggetti Libro, che Spring Boot converte automaticamente in JSON.
- La sicurezza è gestita tramite Spring Security con autenticazione JWT.
- Non serve definire un ViewResolver, poiché il formato di output è JSON.

Avvio dell'applicazione

Con Spring Boot, l'avvio avviene tramite una singola classe @SpringBootApplication:

```
@SpringBootApplication
public class Saos2025Application {
   public static void main(String[] args) {
      SpringApplication.run(Saos2025Application.class, args);
   }
}
```

Non è necessario configurare manualmente WebApplicationInitializer o mappature del DispatcherServlet, perché Spring Boot li gestisce automaticamente.

1.4.Spring Boot Entry Point

La differenza principale tra il punto di ingresso (entry point) di un'applicazione Spring tradizionale e quello di un'applicazione Spring Boot riguarda il modo in cui viene avviato il server e inizializzato il contesto dell'applicazione.

Spring tradizionale

In un'applicazione Spring classica:

- Il server legge il file web.xml, dove è configurato il DispatcherServlet.
- Il DispatcherServlet viene istanziato e crea il WebApplicationContext, leggendo un file XML di configurazione (es. WEB-INF/servletName-servlet.xml).

- I bean definiti in questo contesto vengono caricati e messi a disposizione dell'applicazione.

Con Servlet 3+, si può evitare il file web.xml utilizzando un'implementazione di WebApplicationInitializer, che inizializza il contesto tramite configurazione Java (@Configuration).

Spring Boot

Nel caso di Spring Boot, la procedura è molto più semplice e automatizzata:

- Spring Boot utilizza un server web incorporato (come Tomcat, Jetty o Undertow) che viene avviato direttamente dall'applicazione, senza configurazioni manuali di web.xml o DispatcherServlet.
- Il punto di ingresso è una classe con annotazione @SpringBootApplication e un metodo main() che invoca SpringApplication.run(...).
- Spring Boot si occupa automaticamente di:
  - Creare e configurare il DispatcherServlet
  - Associare Servlet, Filter e ServletContextInitializer al server incorporato
  - Effettuare la scansione automatica dei componenti (@RestController, @Service, @Repository, ecc.) all'interno del package principale e dei suoi sotto-package.

Esempio dal progetto SAOS2025:

```
@SpringBootApplication
public class Saos2025Application {
   public static void main(String[] args) {
      SpringApplication.run(Saos2025Application.class, args);
   }
}
```

Distribuzione in un container esterno

Anche se Spring Boot è pensato per essere eseguito come applicazione standalone, è possibile distribuirlo come WAR in un server esterno (es. Tomcat installato a parte).

In questo caso, si estende SpringBootServletInitializer:

```
@SpringBootApplication
public class Saos2025Application extends SpringBootServletInitializer {
   // eventuali configurazioni aggiuntive
}
```

Il server esterno rileverà questa classe e inizializzerà l'applicazione collegando servlet e filtri come avviene nel server incorporato.

1.5. Build e Distribuzione

Infine, vediamo come un'applicazione può essere impacchettata e distribuita. Entrambi questi framework supportano tecnologie di gestione dei pacchetti comuni come Maven e Gradle; tuttavia, questi framework differiscono molto, quando si tratta di distribuzione. Ad esempio, il plug-

in Spring Boot Maven, che fornisce il supporto Spring Boot in Maven, consente di impacchettare jar eseguibili o archivi .war ed eseguire un'applicazione in "sul posto". Alcuni dei vantaggi di Spring Boot rispetto a Spring nel contesto della distribuzione sono:

- Fornisce il supporto del container incorporato;

- Possibilità di eseguire i jar in modo indipendente utilizzando il comando java -jar;

- Opzione per escludere le dipendenze per evitare potenziali conflitti di jar durante la distribuzione in un container esterno;

- Opzione per specificare i profili attivi durante la distribuzione;

- Generazione di porte casuali per test di integrazione.

1.6. Conclusione

Spring è una scelta eccellente per gli sviluppatori per creare applicazioni Java aziendali. Tuttavia, è molto vantaggioso se utilizzato insieme a Spring Boot. Mentre Spring offre agli sviluppatori flessibilità e versatilità, Spring Boot si concentra sulla riduzione della lunghezza e della configurazione del codice, fornendo così agli sviluppatori il metodo più semplice e veloce per creare un'applicazione. I vantaggi aggiuntivi di Spring Boot sono di grande valore in quanto riducono notevolmente i tempi e gli sforzi di sviluppo.

2. Creare un servizio API REST di alta qualità

Progettare un'API REST non significa solo scrivere del codice che scambia dati: serve creare un'interfaccia chiara, logica e facilmente comprensibile anche da chi non l'ha sviluppata. Un'API ben progettata è essenziale per garantire integrazione, facilità d'uso e scalabilità

Le **API** (Application Programming Interface) sono strumenti software che consentono lo scambio di dati tra applicazioni e dispositivi, permettendo la comunicazione tra sistemi diversi e rendendo disponibili solo le funzionalità necessarie.
Una buona progettazione dell'API è fondamentale: un'implementazione corretta favorisce un'integrazione solida, mentre errori strutturali possono compromettere l'intero progetto.

Per sviluppare un'API REST efficace, conviene seguire alcune regole di base:

- conoscere bene i casi d'uso e le applicazioni che la utilizzeranno;
- avere familiarità con i vincoli del protocollo **http**;
- mantenere coerenza con l'architettura e il framework utilizzato;
- gestire con attenzione le modifiche agli endpoint già esposti per evitare incompatibilità.

2.1. Architettura API: equilibrio tra funzionalità e semplicità

La progettazione di un'API deve unire **struttura tecnica** e **facilità d'uso**. Una buona API:

- è semplice da utilizzare per le operazioni di base;
- Ha un modello di progettazione chiaro, così da ridurre il tempo di apprendimento;
- Mantiene coerenza negli endpoint e nei formati, permettendo a uno sviluppatore di riprendere il lavoro anche dopo tempo;
- Include controlli di qualità e strumenti di monitoraggio per individuare e correggere errori.
- Offre un'esperienza d'uso soddisfacente sia per sviluppatori *backend* che *frontend*;

Senza pianificazione e test accurati, il ciclo di sviluppo rischia di allungarsi e aumentare il numero di bug. Una buona API è vantaggiosa per entrambe le parti:

- **Consumatori dell'API**: meno errori, integrazione più rapida.
- **Fornitori dell'API**: meno richieste di supporto e maggiore soddisfazione degli utenti.

2.2. Tre principi fondamentali per la qualità di un'API REST

Comprendere a fondo il protocollo HTTP e le sue potenzialità (metodi GET, POST, PUT, DELETE, gestione delle risposte, caching, ecc.).

1. Saper collegare in modo logico le risorse ai rispettivi endpoint.
2. Stabilire regole chiare su come rappresentare una risorsa singola o una collezione di risorse.

Seguire le **best practice** e, se possibile, usare framework consolidati aiuta a mantenere qualità e produttività elevate.

2.3. Regole generali di progettazione di un'API REST

Gli endpoint devono essere **intuitivi**: i nomi rappresentano le risorse, i verbi (impliciti nei metodi HTTP) indicano l'azione:

- esempio: GET /libri → ottiene tutti i libri, POST /libri → inserisce un nuovo libro;
- restituire sempre uno stato chiaro (HTTP status code) e, se possibile, un messaggio descrittivo;
- la sintassi e la struttura dei parametri devono essere coerenti e documentate;
- gestire in modo dettagliato le condizioni di errore, spiegando chiaramente la causa.

2.4. Importanza di una buona documentazione

Una documentazione chiara e completa è essenziale per favorire l'adozione dell'API. Deve includere:

- **descrizione del servizio** e del suo scopo;
- **URL degli endpoint**;
- **parametri** accettati e loro significato;
- **esempi di richieste e risposte** (sia in caso di successo che di errore);
- **significato degli status HTTP** usati dall'API.

Strumenti utili per creare documentazione:

- **Swagger** (oggi OpenAPI Specification) – genera documentazione interattiva e codice a partire dalla definizione dell'API;
- **GitHub Wikis** – documentazione collaborativa;
- **Slate** – documentazione elegante e statica.

Swagger è particolarmente utile anche per il lavoro in team, poiché consente a backend e frontend di lavorare sulla stessa base di specifiche.

2.5. Conclusioni

Nel panorama dello sviluppo software moderno, le API RESTful sono diventate uno standard per l'integrazione tra sistemi.
La loro capacità di **astrarre e standardizzare** la comunicazione tra componenti consente alle aziende di:

- Sviluppare più rapidamente.
- Mantenere maggiore flessibilità.
- Integrare più facilmente nuovi servizi.

Oggi, quasi tutte le applicazioni di successo espongono API pubbliche o private per consentire a partner, clienti e sviluppatori di sfruttare al meglio i dati e le funzionalità offerte.

3. CASE STUDY

L'obiettivo di questo progetto è implementare un sistema di **sicurezza e autenticazione** basato su **JWT (JSON Web Token)** all'interno di un servizio REST API sviluppato con **Java Spring Boot**. L'obiettivo finale è testarne e validarne il corretto funzionamento in un contesto reale.

La maggior parte delle applicazioni moderne segue l'architettura ***client-server****: il* ***frontend*** (client) comunica con il ***backend*** (server) per gestire il flusso dati. Questa comunicazione avviene tipicamente tramite **HTTP** e spesso si affida ad **API REST**.

3.1. Prerequisiti

Per creare un servizio API REST con Spring Boot e Java, sono necessari:

- **IDE** (Eclipse, IntelliJ IDEA o anche un semplice editor di testo avanzato);
- **JDK 1.8** o superiore;
- **Maven 3.0** o superiore.

3.2. Creazione del progetto Spring Boot

Il modo più semplice per avviare il progetto è utilizzare **Spring Initializr** (https://start.spring.io/), configurando:

- **Build Tool:** Maven;
- **Linguaggio:** Java;
- **Versione Spring Boot:** scelta desiderata.

Dipendenze consigliate:

- **Spring Data JPA** – per la gestione della persistenza con Hibernate;
- **Spring Web** – per esporre API REST;
- **Spring Boot DevTools** – per facilitare lo sviluppo;
- **MariaDB Driver** (o driver per il DB scelto).

Dopo aver configurato, cliccando su **"Generate"** si otterrà un archivio ZIP contenente la struttura base del progetto, pronto per essere importato nell'IDE.

3.3. Configurazione accesso al database

Spring Boot semplifica la connessione al database tramite il file application.properties (in /src/main/resources).
Esempio di configurazione per MariaDB:

```
spring.datasource.url=jdbc:mariadb://localhost:3306/saos?useSSL=false
spring.datasource.username=root
spring.datasource.password=saos

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MariaDB10Dialect
spring.jpa.hibernate.ddl-auto=update
```

3.4.Creazione della classe Entity

L'entità rappresenta il **modello di dominio** della nostra applicazione. Esempio per la tabella libri:

```
@Entity
@Table(name = "libri")
public class Libro {
   @Id
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   private long id;

   private String nome;
   private String autore;

   // Getters e setters
}
```

- @Entity → definisce una classe come entità JPA.
- @Table → specifica il nome della tabella nel database.
- @Id → identifica la chiave primaria.
- @GeneratedValue → definisce la strategia di generazione dell'ID.

3.5.Creazione della classe Repository

Il repository permette di eseguire operazioni CRUD senza scrivere query SQL.

```
@Repository
public interface LibroRepository extends JpaRepository<Libro, Long> {}
```

JpaRepository fornisce metodi come findAll(), findById(), save(), deleteById().

3.6.Creazione del Rest Controller e delle API

Il controller espone gli endpoint REST e gestisce le richieste HTTP:

```
@RestController
@RequestMapping("/api/libro")
public class LibroController {

   @Autowired
   private LibroRepository libroRepository;

   @GetMapping
   public List<Libro> findAllBooks() {
      return libroRepository.findAll();
   }

   @GetMapping("/{id}")
   public ResponseEntity<Libro> findBookById(@PathVariable long id) {
      return libroRepository.findById(id)
         .map(ResponseEntity::ok)
         .orElse(ResponseEntity.notFound().build());
   }

   @PostMapping
   public Libro saveBook(@Validated @RequestBody Libro libro) {
```

```
        return libroRepository.save(libro);
    }
}
```

Endpoint disponibili:

- **GET** /api/libro → restituisce la lista di tutti i libri
- **GET** /api/libro/{id} → restituisce un libro per ID
- **POST** /api/libro → inserisce un nuovo libro

3.7. Build e avvio del progetto

Per avviare l'applicazione:

```
mvn spring-boot:run
```

oppure, dopo il packaging:

```
mvn package
java -jar target/SAOS2025-0.0.1-SNAPSHOT.jar
```

Se il progetto parte correttamente, sarà accessibile su:

http://localhost:8080/

I log mostreranno l'avvio corretto del contesto Spring Boot e del **DispatcherServlet**.

4. Proteggere le API REST Spring Boot con autenticazione JWT

La **sicurezza** è uno degli aspetti fondamentali nello sviluppo di applicazioni web. **Spring Security**, integrato con **JWT (JSON Web Token)**, è un'ottima soluzione per gestire autenticazione e autorizzazione in un'architettura REST moderna.

L'obiettivo è mettere in sicurezza le API REST del progetto SAOS2025, proteggendo le risorse sensibili e consentendo l'accesso solo agli utenti autorizzati.

4.1.JWT: definizione e funzionamento

Un **JWT** è un token in formato JSON, composto da tre parti:

- ***Header*** – contiene informazioni sul tipo di token e sull'algoritmo di firma;
- ***Payload*** – include i dati (*claims*) come username, ruoli, data di scadenza;
- ***Signature*** – la firma generata con una chiave segreta.

Il flusso di utilizzo è il seguente:

1. il ***client*** invia credenziali di accesso (username e password) a un'API di login;
2. il ***server*** verifica le credenziali e, in caso di esito positivo, genera un token JWT firmato con una chiave segreta;
3. il ***client*** memorizza il token e lo invia in ogni richiesta protetta, nell'*header* ***Authorization*** come *Bearer <token>*;
4. il **server** valida la firma, controlla la scadenza ed i permessi dell'utente prima di rispondere.

Se la validazione fallisce, viene restituito **HTTP 403 Forbidden**.

Proprietà da aggiungere in application.properties:

```
CatDam.SAOS2025.app.jwtSecret= sRSecretKey
```

CatDam.SAOS2025.app.jwtExpirationMs= 86400000

4.2.Creazione dei modelli/entità

Per implementare un sistema di autenticazione e autorizzazione basato su **JWT** in **Spring Boot**, il primo passo è definire il **modello dati**.

In particolare, ci servono:

- ***Utente*** – rappresenta l'account registrato nel sistema (*username, e-mail, password* e ruoli associati);
- ***Ruolo*** – identifica un insieme di permessi (es. UTENTE, MODERATORE, ADMIN);
- ***Relazione molti-a-molti*** – un utente può avere più ruoli e un ruolo può essere associato a più utenti.

Questa relazione è gestita da una **tabella di join** chiamata utente_ruoli.
Le entità sono annotate con **JPA** (@Entity, @Table) per permettere a Hibernate di generare le tabelle in automatico.

L'uso dell'**enum** RuoloEnum semplifica la gestione dei ruoli rendendo più sicuro il codice (evitando errori di battitura nei nomi dei ruoli).

Esempio Utente.java e Ruolo.java:

```
@Entity
@Table(name = "utenti",
      uniqueConstraints = {
            @UniqueConstraint(columnNames = "username"),
            @UniqueConstraint(columnNames = "email")
      })
public class Utente {
   @Id
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   private Long id;
   @NotBlank
   @Size(max = 20)
   private String username;
   @NotBlank
   @Size(max = 50)
   @Email
   private String email;
   @NotBlank
   @Size(max = 120)
   private String password;
   @ManyToMany(fetch = FetchType.LAZY)
   @JoinTable(    name = "utente_ruoli",
         joinColumns = @JoinColumn(name = "id_utente"),
         inverseJoinColumns = @JoinColumn(name = "id_ruolo"))
   private Set roles = new HashSet<>();
   public Utente() {
   }
   public Utente(String username, String email, String password) {
      this.username = username;
      this.email = email;
      this.password = password;
   }

  //getter e setter
}
@Entity
@Table(name = "ruoli")
public class Ruolo {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Integer id;
    @Enumerated(EnumType.STRING)
    @Column(length = 20)
    private RuoloEnum name;
    public Ruolo() {
    }
    public Ruolo(RuoloEnum name) {
        this.name = name;
    }
 //getter e setter
public enum RuoloEnum {
    UTENTE,
    MODERATORE,
    ADMIN,
}
```

4.3.Implementazione repository

I **repository** sono componenti che si occupano di comunicare con il database. In **Spring Data JPA**, estendendo JpaRepository si ottengono automaticamente metodi CRUD come save(), findById(), findAll() senza dover scrivere query SQL.

Nel nostro caso:

- **UtenteRepository** fornisce metodi per cercare un utente per **username** o **email**;
- **RuoloRepository** fornisce metodi per recuperare i ruoli tramite il loro nome (RuoloEnum).

Abbiamo anche aggiunto query personalizzate (@Query) per eventuali esigenze specifiche, sebbene per la maggior parte dei casi JpaRepository sia sufficiente.

```
@Repository
public interface UtenteRepository extends JpaRepository {
    @Query(value = "SELECT * FROM utenti WHERE username LIKE %?1%", nativeQuery = true)
    Optional cercaPerUsername(String username);

    @Query(value = "SELECT * FROM utenti WHERE email LIKE %?1%", nativeQuery = true)
    Optional cercaPerEmail(String email);
}

@Repository
public interface RuoloRepository extends JpaRepository {
    @Query(value = "SELECT * FROM ruoli WHERE name LIKE %?1%", nativeQuery = true)
    Optional cercaPerNome(RuoloEnum name);
    @Query(value = "SELECT * FROM ruoli WHERE name LIKE %?1%", nativeQuery = true)
    List cercaPerNomeList(RuoloEnum name);
}
```

4.4.La classe JwtUtils – gestione token

JwtUtils è una **classe di utilità** che si occupa di:

- **Generare un token JWT** quando un utente si autentica correttamente;
- **Estrarre lo username** dal token;
- **Validare il token** verificando la firma e la scadenza.

Il token viene generato con la libreria **jjwt** e firmato utilizzando un algoritmo di crittografia (es. **HS512**).
Le proprietà jwtSecret e jwtExpirationMs sono configurabili in application.properties, così da poter cambiare

la chiave segreta o la durata del token senza modificare il codice.

```
@Component
public class JwtUtils {
   @Value("${CatDam.SAOS2025.app.jwtSecret}")
   private String jwtSecret;

   @Value("${CatDam.SAOS2025.app.jwtExpirationMs}")
   private int jwtExpirationMs;

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

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

   public boolean validateJwtToken(String authToken) {
      try {
         Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(authToken);
         return true;
      } catch (JwtException e) {
         return false;
      }
   }
}
```

4.5.La classe AuthTokenFilter– filtro di autorizzazione

AuthTokenFilter è un **filtro** che intercetta tutte le richieste HTTP. Il suo compito è:

1. Leggere l'header **Authorization;**
2. Estrarre il token (schema **Bearer**);
3. Validarlo tramite JwtUtils;
4. Recuperare i dettagli dell'utente (DettagliUtenteService);
5. Popolare il **SecurityContext** di Spring Security con l'utente autenticato.

Se il token è assente o non valido, la richiesta prosegue senza impostare un utente autenticato, e se l'endpoint è protetto verrà restituito un **403 Forbidden**.

Questo approccio rende il sistema **stateless**: non memorizziamo sessioni lato server, ma deleghiamo tutta l'autenticazione al token stesso.

```
public class AuthTokenFilter extends OncePerRequestFilter {
   @Autowired
   private JwtUtils jwtUtils;
   @Autowired
   private DettagliUtenteService userDetailsService;
   private static final Logger logger = Logger.getLogger(AuthTokenFilter.class.getName());
```

```
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        try {
            String jwt = parseJwt(request);
            if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
                String username = jwtUtils.getUserNameFromJwtToken(jwt);
                DettagliUtente userDetails = userDetailsService.loadUserByUsername(username);
                UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(
                        userDetails, null, userDetails.getAuthorities());
                authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        } catch (Exception e) {
            logger.log(Level.SEVERE, "Cannot set user authentication: {}", e);
        }
        filterChain.doFilter(request, response);
    }
    private String parseJwt(HttpServletRequest request) {
        String headerAuth = request.getHeader("Authorization");
        if (StringUtils.hasText(headerAuth) && headerAuth.startsWith("Bearer ")) {
            return headerAuth.substring(7, headerAuth.length());
        }
        return null;
    }
}
```

4.6.La classe WebSecurityConfig – configurazione sicurezza

La classe WebSecurityConfig configura **Spring Security** per funzionare con JWT invece che con sessioni tradizionali.

Le operazioni principali sono:

- **Disabilitare CSRF** (.csrf().disable()), perché non usiamo cookie di sessione;
- **Impostare sessioni stateless** (SessionCreationPolicy.STATELESS);
- **Definire endpoint pubblici** con .antMatchers(...).permitAll() (es. login, registrazione, test);
- **Proteggere tutti gli altri endpoint** con .anyRequest().authenticated();
- **Aggiungere il filtro JWT** nella filter chain prima del filtro di autenticazione standard.

Infine, definiamo un PasswordEncoder (BCrypt) per cifrare le password e un AuthenticationManager per gestire l'autenticazione.

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig {
    @Bean
    public AuthTokenFilter authenticationJwtTokenFilter() {
        return new AuthTokenFilter();
    }
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

        http.cors().and().csrf().disable()
.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
                .authorizeRequests().antMatchers("/api/auth/**").permitAll()
                .antMatchers("/api/test/**").permitAll()
                .antMatchers("/api/libro/**").permitAll()
```

```
            .anyRequest().authenticated();
        http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }
    @Bean
    public WebSecurityCustomizer webSecurityCustomizer() {
        return (web) -> web.ignoring().antMatchers("/images/**", "/js/**", "/webjars/**");
    }
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration)
            throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }
}
```

4.7.Payload per login e registrazione

Per facilitare la gestione delle richieste in **JSON**, definiamo due **DTO (Data Transfer Object)**:

- Credenziali – contiene **username** e **password** per il login
- Registrazione – contiene **username**, **e-mail**, **password** e l'elenco dei ruoli da assegnare in fase di creazione dell'account

Questi DTO permettono di validare automaticamente i campi con le annotazioni di **Bean Validation** (@NotBlank, @Size, @Email), riducendo errori e codice ridondante.

```
public class Credenziali {
    @NotBlank
    private String username;

    @NotBlank
    private String password;

   // getter e setter
}
public class Registrazione {
    @NotBlank
    @Size(min = 3, max = 20)
    private String username;

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

    private Set ruoli;

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

  //getter e setter
}
```

4.8. AuthController – gestione autenticazione e registrazione

AuthController espone due endpoint principali:

1. POST /api/auth/login – verifica le credenziali e restituisce:
    - Token JWT
    - Dati utente (id, username, email)
    - Ruoli associati
2. POST /api/auth/registrazione – permette a un **ADMIN** di creare un nuovo utente, controllando che username ed email non siano già presenti nel database.

Durante il login:

- L'AuthenticationManager verifica le credenziali
- Se corrette, JwtUtils genera un token JWT
- Il token viene incluso nella risposta insieme ai dati dell'utente

Durante la registrazione:

- Si controlla se username ed email sono univoci
- La password viene cifrata con **BCrypt**
- I ruoli vengono assegnati in base ai dati ricevuti (o impostati di default a UTENTE)
- L'utente viene salvato nel database

Questo approccio rende l'API sicura, scalabile e facilmente estendibile per aggiungere nuove regole di accesso.

```
@CrossOrigin(origins = "*", maxAge = 3600)

@RestController
@RequestMapping("/api/auth")
public class AuthController{

   @Autowired
   private UtenteRepository userRepo;
   @Autowired private JwtUtils jwtUtil;
   @Autowired private AuthenticationManager authManager;
   @Autowired private PasswordEncoder passwordEncoder;
   @Autowired private RuoloRepository roleRepository;

   @PostMapping("/login")
   public ResponseEntity authenticateUser(@Valid @RequestBody Credenziali loginRequest) {
      Authentication authentication = authManager.authenticate(
            new UsernamePasswordAuthenticationToken(loginRequest.getUsername(), loginRequest.getPassword()));
      SecurityContextHolder.getContext().setAuthentication(authentication);
      String jwt = jwtUtil.generateJwtToken(authentication);

      DettagliUtente userDetails = (DettagliUtente) authentication.getPrincipal();
      List roles = userDetails.getAuthorities().stream()
            .map(item -> item.getAuthority())
            .collect(Collectors.toList());
      return ResponseEntity.ok(new JwtResponse(jwt,
            userDetails.getId(),
            userDetails.getUsername(),
```

```
            userDetails.getEmail(),
            roles));
    }

    @PostMapping("/registrazione")
    @PreAuthorize("hasAuthority('ADMIN')")
    public ResponseEntity registerUser(@Valid @RequestBody Registrazione signUpRequest) {

        Optional username = userRepo.cercaPerUsername(signUpRequest.getUsername());
        if (username.isPresent()) {
            return ResponseEntity
                    .badRequest()
                    .body(new MessageResponse("Error: Username già presente!"));
        }

        Optional email = userRepo.cercaPerEmail(signUpRequest.getEmail());
        if (email.isPresent()) {
            return ResponseEntity
                    .badRequest()
                    .body(new MessageResponse("Error: Email già presente!"));
        }
        Utente user = new Utente(signUpRequest.getUsername(),
                signUpRequest.getEmail(),
                passwordEncoder.encode(signUpRequest.getPassword()));
        Set strRoles = signUpRequest.getRuoli();

        Set roles = new HashSet<>();
        if (strRoles == null) {
            Ruolo userRole = roleRepository.cercaPerNome(RuoloEnum.UTENTE)
                    .orElseThrow(() -> new RuntimeException("Error: Role "+RuoloEnum.UTENTE+" is not found"));
            roles.add(userRole);
        } else {
            strRoles.forEach(role -> {
                switch (role) {
                    case "ADMIN":
                        Ruolo adminRole = roleRepository.cercaPerNome(RuoloEnum.ADMIN)
                                .orElseThrow(() -> new RuntimeException("Error: Role "+RuoloEnum.ADMIN+" is not found"));
                        roles.add(adminRole);
                        break;
                    case "MODERATORE":
                        Ruolo modRole = roleRepository.cercaPerNome(RuoloEnum.MODERATORE)
                                .orElseThrow(() -> new RuntimeException("Error: Role "+RuoloEnum.MODERATORE+" is not
found"));
                        roles.add(modRole);
                        break;
                    default:
                        Ruolo userRole = roleRepository.cercaPerNome(RuoloEnum.UTENTE)
                                .orElseThrow(() -> new RuntimeException("Error: Role "+RuoloEnum.UTENTE+" is not found"));
                        roles.add(userRole);
                }
            });
        }
        user.setRoles(roles);
        userRepo.save(user);
        return ResponseEntity.ok(new MessageResponse("Utente registrato correttamente!"));
    }
}
```