

Guida Rapida a Spring Boot (Dark Edition)

November 25, 2025

INDICE

1	Creazione di un'API REST	3
1.1	Struttura generale dei componenti	3
1.2	Esempio di Entity	4
1.3	Repository	4
1.4	Service	4
1.5	HTTP Controller	4
1.6	Gestione delle Eccezioni	5
1.7	Logging interno	5
1.8	Data Transfer Object (DTO)	6
1.8.1	Esempio DTO	6
1.9	Validazione dei dati	6
1.9.1	Annotazioni comuni	6
1.9.2	Esempio Controller con validazione	6
1.9.3	Gestione errori di validazione	6
1.10	Relazioni tra Entity	7
1.10.1	One-to-One	7
1.10.2	One-to-Many	7
1.10.3	Many-to-Many	7
1.11	Mapping tra DTO ed Entity	8
1.11.1	Esempio mapping manuale	8
1.11.2	Esempio mapping automatico con MapStruct	8
1.12	Controller e API REST	9
1.12.1	Principali annotazioni Spring Web	9
2	Gestione delle Entità con Spring Data JPA	9
2.1	Creazione di un'Entity	9
2.2	Repository Base	10
2.3	Query Personalizzate	10
2.4	Relazioni tra Entity	10
2.5	Transazioni e gestione logica	10
2.6	Paginazione e Ordinamento	11
2.7	Custom Repository Implementation	11
2.8	Logica avanzata con Specification e Criteria API	11
2.9	Uso delle Specification nel Service	12
2.10	Entity, Repository e JPA	12
2.10.1	Definizione e annotazioni principali	12
3	Integrazione di Spring Security	12
3.1	Filtro JWT personalizzato con Firebase	12
3.2	Configurazione SecurityFilterChain	13
3.3	Gestione dei ruoli e permessi	14
3.4	Autenticazione JWT interno	14
3.5	Logging e gestione eccezioni	14
3.6	Considerazioni finali	14
3.7	CustomUserDetailsService: integrazione con Spring Security	14
3.7.1	Classe di esempio	15
3.7.2	Spiegazione	15
3.7.3	Come si collega al SecurityConfig	15
3.8	JwtInternalAuthFilter: gestione dei token interni e Firebase	15

Introduzione

Spring Boot è un framework che semplifica in modo estremo lo sviluppo di applicazioni Java basate su Spring. L'obiettivo principale è ridurre al minimo la configurazione manuale, fornendo una serie di default intelligenti e un ecosistema modulare che permette di costruire API REST, gestire la sicurezza e interfacciarsi con database in modo rapido e consistente.

A livello macroscopico, Spring Boot offre:

- **Auto-configuration:** rileva automaticamente le dipendenze e configura i componenti principali.
- **Starter dependencies:** pacchetti ottimizzati che includono tutto ciò che serve per funzionalità specifiche.
- **Embedded server:** Tomcat, Jetty o Undertow già integrati per l'esecuzione immediata.
- **Integrazione fluida con Spring Data, Spring Security e altri moduli Spring.**
- **Facilità di creazione di API REST** tramite controller e annotazioni intuitive.

Prima di proseguire, è utile chiarire due concetti fondamentali del mondo Spring:

Dependency Injection (DI)

La Dependency Injection è un pattern che permette di delegare a Spring la creazione e la gestione delle dipendenze tra gli oggetti. Invece di istanziare manualmente le classi, è il container Spring a fornirle già pronte, semplificando il codice e migliorando la testabilità.

Inversion of Control (IoC)

L'Inversion of Control è il principio secondo cui non è il codice dell'applicazione a controllare il flusso di creazione degli oggetti, ma è il container Spring a gestire tutto. L'applicazione quindi "cede il controllo" al framework, ottenendo configurazioni più pulite e modulari.

Nei capitoli seguenti verranno aggiunte sezioni dedicate alle operazioni più comuni nello sviluppo con Spring Boot.

1 Creazione di un'API REST

In questa sezione viene mostrato come costruire un'API REST completa con Spring Boot, includendo entità, repository, servizio, controller, gestione delle eccezioni, logging e risposta HTTP strutturata.

1.1 Struttura generale dei componenti

Una tipica API Spring Boot segue questa architettura:

- **Entity:** rappresenta la tabella del database.
- **Repository:** interfaccia per l'accesso ai dati tramite Spring Data JPA.
- **Service:** contiene la logica applicativa.
- **Controller:** espone le API REST.
- **Exception Handler:** gestisce errori e risposte HTTP custom.
- **Log:** utile per debug e monitoraggio.

1.2 Esempio di Entity

```
@Entity public class User { @Id @GeneratedValue(strategy = GenerationType.IDENTITY) private Long id;  
private String name;  
private String email;  
  
// getters e setters  
}
```

1.3 Repository

```
public interface UserRepository extends JpaRepository<User, Long> { Optional<User> findByEmail(String ema
```

1.4 Service

```
@Service public class UserService {  
private final UserRepository repo;  
private static final Logger log = LoggerFactory.getLogger(UserService.class);  
  
public UserService(UserRepository repo) {  
    this.repo = repo;  
}  
  
public User create(User u) {  
    log.info("Creazione utente: {}", u.getEmail());  
    return repo.save(u);  
}  
  
public User getById(Long id) {  
    return repo.findById(id)  
        .orElseThrow(() -> new NotFoundException("Utente non trovato"));  
}  
  
public List<User> getAll() {  
    return repo.findAll();  
}  
  
public User update(Long id, User u) {  
    User existing = getById(id);  
    existing.setName(u.getName());  
    existing.setEmail(u.getEmail());  
    return repo.save(existing);  
}  
  
public void delete(Long id) {  
    repo.delete(getById(id));  
}
```

1.5 HTTP Controller

```
@RestController @RequestMapping("/api/users") public class UserController {  
private final UserService service;
```

```

public UserController(UserService service) {
    this.service = service;
}

@PostMapping
public ResponseEntity<User> create(@RequestBody User user) {
    return ResponseEntity.status(HttpStatus.CREATED).body(service.create(user));
}

@GetMapping("/{id}")
public ResponseEntity<User> get(@PathVariable Long id) {
    return ResponseEntity.ok(service.getById(id));
}

@GetMapping
public ResponseEntity<List<User>> getAll() {
    return ResponseEntity.ok(service.getAll());
}

@PutMapping("/{id}")
public ResponseEntity<User> update(@PathVariable Long id, @RequestBody User user) {
    return ResponseEntity.ok(service.update(id, user));
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> delete(@PathVariable Long id) {
    service.delete(id);
    return ResponseEntity.noContent().build();
}
}

```

1.6 Gestione delle Eccezioni

```

@ResponseStatus(HttpStatus.NOT_FOUND) public class NotFoundException extends RuntimeException { public N
@ExceptionHandler(NotFoundException.class)
public ResponseEntity<Map<String, Object>> handleNotFound(NotFoundException ex) {
    Map<String, Object> body = new HashMap<>();
    body.put("error", ex.getMessage());
    body.put("timestamp", LocalDateTime.now());
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(body);
}

@ExceptionHandler(Exception.class)
public ResponseEntity<Map<String, Object>> handleGeneric(Exception ex) {
    Map<String, Object> body = new HashMap<>();
    body.put("error", "Errore interno del server");
    body.put("details", ex.getMessage());
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(body);
}
}

```

1.7 Logging interno

Spring Boot usa SLF4J + Logback. Per loggare:

```
private static final Logger log = LoggerFactory.getLogger(NomeClasse.class); log.info("Messaggio informativo")
```

1.8 Data Transfer Object (DTO)

I DTO (Data Transfer Object) sono classi utilizzate per trasferire dati tra i vari livelli dell'applicazione. Servono a:

- Separare i modelli interni (Entity) dalla rappresentazione esterna esposta tramite API.
- Evitare di esporre campi sensibili o non necessari.
- Validare correttamente i dati in ingresso.
- Ridurre gli accoppiamenti e rendere più semplice cambiare struttura interna senza rompere il contratto dell'API.

1.8.1 Esempio DTO

```
public class UserDTO {  
    @NotBlank(message = "Username obbligatorio")  
    private String username;  
  
    @Email(message = "Email non valida")  
    private String email;  
}
```

1.9 Validazione dei dati

Spring utilizza la libreria Jakarta Validation per applicare vincoli ai campi dei DTO e delle Entity.

1.9.1 Annotazioni comuni

- `@NotBlank` – il campo non può essere vuoto.
- `@Email` – deve essere un indirizzo email valido.
- `@Size(min, max)` – lunghezza minima e massima.
- `@Positive` – deve essere maggiore di zero.
- `@Past / @Future` – date temporali.

1.9.2 Esempio Controller con validazione

```
@PostMapping("/users")  
public ResponseEntity<UserDTO> create(@Valid @RequestBody UserDTO dto) {  
    return ResponseEntity.ok(service.createUser(dto));  
}
```

1.9.3 Gestione errori di validazione

```
@RestControllerAdvice  
public class GlobalExceptionHandler {  
  
    @ExceptionHandler(MethodArgumentNotValidException.class)  
    public ResponseEntity<?> handleValidation(MethodArgumentNotValidException ex) {  
        Map<String, String> errors = new HashMap<>();  
        ex.getBindingResult().getFieldErrors().forEach(err ->  
            errors.put(err.getField(), err.getDefaultMessage())  
        );  
        return ResponseEntity.badRequest().body(errors);  
    }  
}
```

1.10 Relazioni tra Entity

Le relazioni definiscono come le entità sono collegate nel database. Spring Data JPA usa le annotazioni di JPA per descriverle.

1.10.1 One-to-One

Relazione uno a uno: un record è collegato ad un solo altro record.

```
@Entity
public class UserProfile {

    @Id @GeneratedValue
    private Long id;

    private String bio;

    @OneToOne
    @JoinColumn(name="user_id")
    private User user;
}
```

1.10.2 One-to-Many

Relazione uno-a-molti: una entità possiede una lista di entità correlate.

```
@Entity
public class User {

    @Id @GeneratedValue
    private Long id;

    @OneToMany(mappedBy="user", cascade=CascadeType.ALL)
    private List<Post> posts = new ArrayList<>();
}

@Entity
public class Post {

    @Id @GeneratedValue
    private Long id;

    @ManyToOne
    private User user;
}
```

1.10.3 Many-to-Many

Relazione molti-a-molti con tabella ponte.

```
@Entity
public class Student {

    @Id @GeneratedValue
    private Long id;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name="student_id"),
        inverseJoinColumns = @JoinColumn(name="course_id")
    )
    private Set<Course> courses;
}
```

1.11 Mapping tra DTO ed Entity

È importante separare DTO ed Entity. La conversione può essere fatta:

- Manualmente tramite metodi dedicati.
- Automaticamente tramite MapStruct.

1.11.1 Esempio mapping manuale

```
public User toEntity(UserDTO dto) {  
    User u = new User();  
    u.setUsername(dto.getUsername());  
    u.setEmail(dto.getEmail());  
    return u;  
}  
  
public UserDTO toDTO(User u) {  
    UserDTO dto = new UserDTO();  
    dto.setUsername(u.getUsername());  
    dto.setEmail(u.getEmail());  
    return dto;  
}
```

1.11.2 Esempio mapping automatico con MapStruct

MapStruct genera automaticamente i metodi per convertire Entity in DTO e viceversa, eliminando il codice manuale.

Mapper Interface

```
@Mapper(componentModel = "spring")  
public interface UserMapper {  
  
    UserDTO toDTO(User user);  
  
    User toEntity(UserDTO dto);  
  
    List<UserDTO> toDTOList(List<User> users);  
}
```

Uso nel Service

```
@Service  
public class UserService {  
  
    private final UserRepository repo;  
    private final UserMapper mapper;  
  
    public UserService(UserRepository repo, UserMapper mapper) {  
        this.repo = repo;  
        this.mapper = mapper;  
    }  
  
    public UserDTO getUser(Long id) {  
        User user = repo.findById(id)  
            .orElseThrow(() -> new RuntimeException("User non trovato"));  
        return mapper.toDTO(user);  
    }  
  
    public UserDTO createUser(UserDTO dto) {  
        User user = mapper.toEntity(dto);  
        User saved = repo.save(user);  
    }
```

```

        return mapper.toDTO(saved);
    }
}

```

Vantaggi

- Niente codice boilerplate manuale
- Conversione veloce e sicura a compile-time
- Supporta anche liste e mappature complesse

1.12 Controller e API REST

1.12.1 Principali annotazioni Spring Web

- `@RestController`: combina `@Controller` e `@ResponseBody`.
- `@RequestMapping`: mappa un path base per tutte le rotte del controller.
- `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`: mappano i metodi HTTP specifici.
- `@PathVariable`: estrae variabili dai path.
- `@RequestParam`: estrae parametri query string.
- `@RequestBody`: deserializza il body JSON in oggetti Java.

2 Gestione delle Entità con Spring Data JPA

Spring Data JPA semplifica enormemente l'accesso ai database relazionali tramite JPA. Offre repository preconfigurati, query derivata dai nomi dei metodi, e supporta transazioni, paginazione, sorting e relazioni complesse.

2.1 Creazione di un'Entity

```

@Entity
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(length = 500)
    private String description;

    private Double price;

    @Temporal(TemporalType.TIMESTAMP)
    private Date createdAt = new Date();

    // getters e setters
}

```

2.2 Repository Base

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
  
    // Query derivata: trova prodotti per nome  
    List<Product> findByName(String name);  
  
    // Query derivata con condizioni  
    List<Product> findByPriceGreaterThanOrEqual(Double price);  
  
    // Ordinamento automatico  
    List<Product> findAllByOrderByIdDesc();  
}
```

2.3 Query Personalizzate

```
public interface ProductRepository extends JpaRepository<Product, Long> {  
  
    @Query("SELECT p FROM Product p WHERE p.price > :minPrice")  
    List<Product> findExpensiveProducts(@Param("minPrice") Double minPrice);  
  
    // Native query  
    @Query(value = "SELECT * FROM product WHERE price < ?1", nativeQuery = true)  
    List<Product> findCheapProducts(Double price);  
}
```

2.4 Relazioni tra Entity

```
// One-to-Many: un Category ha molti Products  
@Entity  
public class Category {  
    @Id @GeneratedValue  
    private Long id;  
  
    private String name;  
  
    @OneToMany(mappedBy = "category", cascade = CascadeType.ALL)  
    private List<Product> products = new ArrayList<>();  
}  
  
// Many-to-One nella Product  
@ManyToOne  
@JoinColumn(name = "category_id")  
private Category category;
```

2.5 Transazioni e gestione logica

```
@Service  
@Transactional  
public class ProductService {  
  
    private final ProductRepository repo;
```

```

public ProductService(ProductRepository repo) {
    this.repo = repo;
}

public Product create(Product p) {
    return repo.save(p);
}

public void updatePrice(Long id, Double newPrice) {
    Product p = repo.findById(id).orElseThrow(() -> new RuntimeException("Prodotto non trovato"));
    p.setPrice(newPrice);
    // la transazione fa l'update automaticamente al commit
}

public void deleteProduct(Long id) {
    repo.deleteById(id);
}
}

```

2.6 Paginazione e Ordinamento

```

Pageable pageable = PageRequest.of(0, 10, Sort.by("price").descending());
Page<Product> page = repo.findAll(pageable);

page.getContent().forEach(p -> System.out.println(p.getName()));

```

2.7 Custom Repository Implementation

```

public interface ProductRepositoryCustom {
    List<Product> findTopSellingProducts(int limit);
}

public class ProductRepositoryImpl implements ProductRepositoryCustom {

    @PersistenceContext
    private EntityManager em;

    @Override
    public List<Product> findTopSellingProducts(int limit) {
        return em.createQuery("SELECT p FROM Product p ORDER BY p.sales DESC", Product.class)
            .setMaxResults(limit)
            .getResultList();
    }
}

```

2.8 Logica avanzata con Specification e Criteria API

```

public class ProductSpecifications {

    public static Specification<Product> hasMinPrice(Double price) {
        return (root, query, cb) -> cb.greaterThan(root.get("price"), price);
    }

    public static Specification<Product> hasName(String name) {
        return (root, query, cb) -> cb.like(cb.lower(root.get("name")), "%" + name.toLowerCase() + "%");
    }
}

```

```
    }
}
```

2.9 Uso delle Specification nel Service

```
List<Product> results = repo.findAll(
    Specification.where(ProductSpecifications.hasMinPrice(100.0))
        .and(ProductSpecifications.hasName("laptop")))
);
```

2.10 Entity, Repository e JPA

2.10.1 Definizione e annotazioni principali

- `@Entity`: definisce una classe come entità JPA mappata su una tabella del database.
- `@Table(name="nome_tabella")`: opzionale, permette di specificare il nome della tabella.
- `@Id`: identifica la chiave primaria dell'entità.
- `@GeneratedValue(strategy = GenerationType.IDENTITY)`: definisce la strategia di generazione automatica dell'ID.
- `@Column`: permette di configurare nome, lunghezza, nullable, unique, ecc.
- `@OneToMany`, `@ManyToOne`, `@OneToOne`, `@ManyToMany`: definiscono relazioni tra entità.
- `@JoinColumn`: specifica la colonna di join per relazioni.

3 Integrazione di Spring Security

Spring Security è il framework di riferimento per la gestione di autenticazione, autorizzazione, sicurezza HTTP e gestione dei token JWT. In questa guida, integriamo sia JWT generati internamente da Spring che JWT Firebase.

3.1 Filtro JWT personalizzato con Firebase

Il filtro estende `OncePerRequestFilter` e intercetta ogni richiesta per verificare il token Firebase:

```
@Component
public class FirebaseJwtFilter extends OncePerRequestFilter {

    private final FirebaseAuth firebaseAuth;
    private static final Logger logger = LoggerFactory.getLogger(FirebaseJwtFilter.class);

    public FirebaseJwtFilter(FirebaseAuth firebaseAuth) {
        this.firebaseioAuth = firebaseAuth;
    }

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

        String token = extractToken(request);
        logger.info("Token trovato: " + (token != null ? "Sì" : "No"));

        if (token != null) {
            try {
```

```

        FirebaseToken decodedToken = firebaseAuth.verifyIdToken(token);
        String uid = decodedToken.getUid();
        List<GrantedAuthority> authorities = Collections.singletonList(new SimpleGrantedAuthority(""));

        UsernamePasswordAuthenticationToken authentication =
            new UsernamePasswordAuthenticationToken(uid, null, authorities);
        authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
        SecurityContextHolder.getContext().setAuthentication(authentication);

    } catch (FirebaseAuthException e) {
        response.setStatus(HttpServletRequest.SC_UNAUTHORIZED);
        response.getWriter().write("Unauthorized: Firebase token invalid or expired");
        return;
    }
}

filterChain.doFilter(request, response);
}

private String extractToken(HttpServletRequest request) {
    String header = request.getHeader("Authorization");
    if (header != null && header.startsWith("Bearer ")) {
        return header.substring(7);
    }
    return null;
}
}

```

3.2 Configurazione SecurityFilterChain

```

@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    private final FirebaseAuth firebaseAuth;
    private static final Logger logger = LoggerFactory.getLogger(SecurityConfig.class);

    @Bean
    public FirebaseJwtFilter firebaseAuthenticationFilter() {
        return new FirebaseJwtFilter(firebaseAuth);
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        logger.info("Configurazione della sicurezza HTTP...");
        http
            .csrf(csrf -> csrf.disable())
            .authorizeRequests(requests -> requests
                .requestMatchers("/auth/**", "/validate").permitAll()
                .anyRequest().authenticated()
            )
            .sessionManagement(management -> management
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )
            .addFilterBefore(new JwtInternalAuthFilter(), UsernamePasswordAuthenticationFilter.class)
            .addFilterBefore(firebaseAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class)
            .cors(withDefaults());
    }
}

```

```

        logger.info("Configurazione della sicurezza completata.");
        return http.build();
    }
}

```

3.3 Gestione dei ruoli e permessi

- ROLE_USER: accesso standard per utenti autenticati
- ROLE_ADMIN: privilegi amministrativi
- Personalizzazione dei ruoli può essere fatta direttamente nel filtro FirebaseJWT o nel JWT interno

3.4 Autenticazione JWT interno

- JWT generato da Spring può essere gestito da un filtro separato (`JwtInternalAuthFilter`)
- Validazione token, estrazione userId, e settaggio di `SecurityContextHolder`
- Possibile combinare i due JWT (Firebase + interno) per flussi ibridi

3.5 Logging e gestione eccezioni

- `LoggerFactory.getLogger()` per logging dettagliato
- Logging di richieste, token estratti e fallimenti di autenticazione
- Risposte HTTP 401 in caso di token mancante o non valido

3.6 Considerazioni finali

- Sessione stateless (`SessionCreationPolicy.STATELESS`)
- Permette di combinare autenticazione interna + Firebase
- Tutti i filtri vengono aggiunti prima di `UsernamePasswordAuthenticationFilter`
- Endpoint pubblici (/auth, /validate) rimangono liberamente accessibili

3.7 CustomUserDetailsService: integrazione con Spring Security

Spring Security utilizza l'interfaccia `UserDetailsService` per caricare le informazioni dell'utente durante il processo di autenticazione. Creare un servizio personalizzato permette di integrare database propri o sistemi esterni, anche quando si usa Firebase o JWT esterni.

3.7.1 Classe di esempio

```
@Component
public class CustomUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        // Recupera l'utente dal database
        UserAccount user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not found"));

        // Costruisce UserDetails usando Spring Security User builder
        return User.builder()
            .username(user.getUsername())
            .password("")           // password vuota perché usa Firebase
            .roles("USER")          // oppure .authorities(...) se preferisci
            .build();
    }
}
```

3.7.2 Spiegazione

- **UserDetailsService**: interfaccia principale per caricare informazioni di autenticazione e autorizzazione.
- **loadUserByUsername**: metodo invocato da Spring Security durante il login, deve restituire un oggetto **UserDetails**.
- **UserDetails**: rappresenta l'utente autenticato con username, password e ruoli/authorities.
- **Integrazione con JWT o Firebase**: anche se la password non viene utilizzata (ad esempio perché si usa Firebase), l'oggetto **UserDetails** permette a Spring Security di gestire il contesto di sicurezza.
- **Ruoli e authorities**: con **.roles("USER")** si assegna il ruolo base, ma si può anche usare **.authorities(...)** per permessi più granulari.

3.7.3 Come si collega al SecurityConfig

- Il **CustomUserDetailsService** viene usato implicitamente da Spring Security quando si configura un **AuthenticationProvider** interno.
- In combinazione con JWT o Firebase, permette di avere informazioni coerenti di ruolo/utente anche senza password locale.

3.8 JwtInternalAuthFilter: gestione dei token interni e Firebase

Questo filtro estende **OncePerRequestFilter** e permette di gestire in modo ibrido:

- JWT generati internamente dall'applicazione (Spring JWT)
- JWT Firebase (controllando la presenza del claim 'kid')

3.8.1 Funzionamento generale

1. Intercetta tutte le richieste HTTP
2. Controlla se l'header **Authorization** contiene un token
3. Se il token contiene il claim 'kid' → considera il token come Firebase e lo verifica tramite **FirebaseAuth**
4. Se il token non contiene 'kid' → lo considera un JWT interno e lo verifica tramite HMAC256
5. Popola il **SecurityContextHolder** con un oggetto **UsernamePasswordAuthenticationToken** valido
6. In caso di errore o token non valido → ritorna HTTP 401

3.8.2 Esempio di filtro

```
public class JwtInternalAuthFilter extends OncePerRequestFilter {

    private UsernamePasswordAuthenticationToken validateJwtInterno(String token) {
        try {
            Algorithm algorithm = Algorithm.HMAC256("CHIAVE_SEGRETA"); // sostituire con la chiave reale
            JWTVerifier verifier = JWT.require(algorithm).build();
            DecodedJWT jwt = verifier.verify(token);

            String userId = jwt.getSubject();
            if (userId == null) return null;

            return new UsernamePasswordAuthenticationToken(userId, null, List.of());
        } catch (JWTVerificationException e) {
            return null;
        }
    }

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

        String authHeader = request.getHeader("Authorization");
        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }

        String token = authHeader.substring(7);

        try {
            DecodedJWT decodedJWT = JWT.decode(token);

            if (decodedJWT.getKeyId() != null) {
                // Token Firebase
                FirebaseAuth firebaseAuth = FirebaseAuth.getInstance().verifyIdToken(token);
                UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken(firebaseAuth.getUid(), null, List.of());
                SecurityContextHolder.getContext().setAuthentication(authentication);
            } else {
                // Token interno
                UsernamePasswordAuthenticationToken authentication = validateJwtInterno(token);
                if (authentication == null) throw new RuntimeException("Token JWT interno non valido");
                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        } catch (FirebaseAuthException | JWTDecodeException e) {
            response.setStatus(HttpServletRequest.SC_UNAUTHORIZED);
            response.getWriter().write("Token non valido");
            return;
        }

        filterChain.doFilter(request, response);
    }
}
```

3.8.3 Punti chiave

- **Validazione ibrida:** consente di gestire sia token Firebase che JWT interni

- **SecurityContextHolder**: imposta l'autenticazione per tutta la request
- **Logging**: utilizzo di SLF4J per monitorare token validi e fallimenti
- **HTTP 401**: restituito quando il token non è presente o non valido
- **Integrazione con SecurityConfig**: aggiunto con `addFilterBefore()` prima di `UsernamePasswordAuthenticationFilter` per garantire la validazione su tutte le API protette

3.8.4 Vantaggi

- Supporto multi-token senza duplicare filtri
- Mantiene lo stato stateless (`SessionCreationPolicy.STATELESS`)
- Permette di avere ruoli coerenti tra JWT interni e Firebase
- Centralizza la logica di autenticazione in un unico filtro

3.8.5 Filtri

- `OncePerRequestFilter`: garantisce che il filtro venga eseguito una sola volta per request.
- `FirebaseJwtFilter`: verifica i token Firebase.
- `JwtInternalAuthFilter`: verifica JWT generati internamente.
- Filtri aggiunti in `SecurityConfig` con `addFilterBefore()`.

3.8.6 SecurityConfig

- `@EnableWebSecurity`: abilita le configurazioni di sicurezza Spring.
- `SecurityFilterChain`: definisce regole HTTP, permessi, CSRF, cors e filtri.
- Endpoint pubblici: `permitAll()` per login, validate, swagger ecc.
- Endpoint protetti: `authenticated()` per tutte le altre richieste.

3.8.7 Lombok e UserDetails

- `CustomUserDetailsService + @Builder, @Data`: semplifica creazione di oggetti UserDetails.
- Permette di non scrivere getter/setter manuali su entità o DTO.

4 CORS (Cross-Origin Resource Sharing)

4.1 Concetto

CORS permette di controllare quali domini esterni possono accedere alle API REST. Utile quando frontend e backend sono su domini diversi.

4.2 Configurazione globale

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
            .allowedOrigins("[http://localhost:3000](http://localhost:3000)")
            .allowedMethods("GET", "POST", "PUT", "DELETE")
            .allowCredentials(true);
    }
}
```

4.3 Configurazione a livello di controller

```
@RestController  
@CrossOrigin(origins = "[http://localhost:3000] (http://localhost:3000)")  
public class UserController { ... }
```

5 Swagger / OpenAPI

5.1 Concetto

Swagger (OpenAPI) permette di documentare automaticamente le API REST e testarle tramite UI interattiva.

5.2 Dipendenze principali (Maven)

```
<dependency> <groupId>org.springdoc</groupId> <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
```

5.3 Annotazioni utili

- `@Operation(summary="...")`: descrive l'endpoint
- `@ApiResponse(responseCode="200", description="...")`: descrive le risposte HTTP
- `@Parameter`: descrive parametri query/path/body

5.4 Accesso alla UI

Una volta configurato, le API sono testabili su: `/swagger-ui.html` o `/swagger-ui/index.html`