Spring Validation Starter

Validazione Dichiarativa in Spring Boot

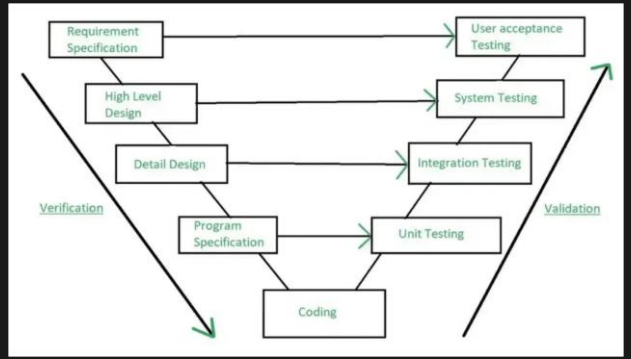
Cos'è Spring Validation Starter

> **La validazione dei dati è fondamentale per la sicurezza e la qualità delle applicazioni.**

Spring Validation Starter è il modulo che fornisce supporto dichiarativo per la validazione dei dati. Basato su **Hibernate Validator**, implementa lo standard **JSR 380 (Bean Validation)**.

Questo approccio permette di definire regole tramite semplici annotazioni, eliminando la necessità di scrivere codice imperativo complesso e rendendo il codice più leggibile e mantenibile.

La dipendenza spring-boot-starter-validation integra automaticamente Hibernate Validator, consentendo di validare oggetti di dominio, parametri di richiesta e entità JPA in modo standardizzato.

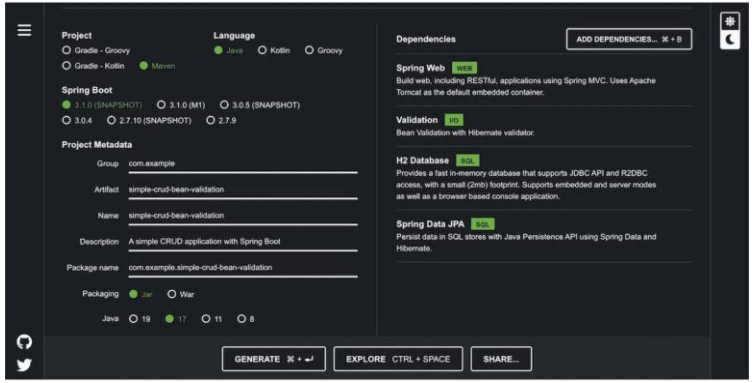


Setup e Dipendenze

Per abilitare la validazione, aggiungere la dipendenza **spring-boot-starter-validation**. Dalla versione 2.3, non è più inclusa di default.

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

Configurazione Automatica: Spring Boot configura automaticamente Hibernate Validator senza passaggi aggiuntivi.



Annotazioni Principali

Parte 1: Validazione di Base

@NotNull

Garantisce che il valore non sia **null**. Accetta comunque stringhe vuote o spazi.

@Size(min=x, max=y)

Verifica che la lunghezza di una String, Collection, Map o Array sia compresa tra i limiti specificati.

@NotBlank

Solo per String. Verifica che non sia null e che la lunghezza trimata sia > 0.

@Min / @Max

Per tipi numerici (o rappresentazioni stringa di numeri). Verifica che il valore sia >= o <= del limite.

@NotEmpty

Verifica che il valore non sia null e che la sua dimensione/lunghezza sia > 0.

@Email

Valida che la stringa sia un indirizzo email ben formato (secondo le specifiche standard).

Annotazioni Principali - Parte 2

Annotazioni Avanzate per Scenari Complessi

Strutturali & Metodo

@Valid

Abilita la **validazione ricorsiva**. Fondamentale per validare oggetti annidati e gerarchie complesse.

@Validated

Specifica di Spring. Abilita la validazione a **livello di metodo** (es. nei Service) e supporta i **Validation Groups**.

Vincoli Specifici

@Email	Valida formato email (RFC 5322).
@Pattern	Validazione tramite Regex custom.
@Positive	Solo numeri strettamente positivi.
@PastOrPresent	Date nel passato o oggi.
@Digits	Precisione numerica (interi/fraz).

Implementazione Base - Modello

La validazione inizia dalla definizione del modello. Annotando i campi della classe di dominio definiamo le regole che i dati devono rispettare.

In questo esempio usiamo **@NotBlank** per i campi obbligatori e **@Email** per garantire il formato corretto dell'indirizzo.


```java
@Entity
public class Utente {
     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private long id;
     @NotBlank(message = "il nome è obbligatorio")
     private String nome;
     @NotBlank(message = "l'email è obbligatoria")
     @Email(message = "l'email deve essere valida")
     private String email;
     // Getters and Setters...
}
```

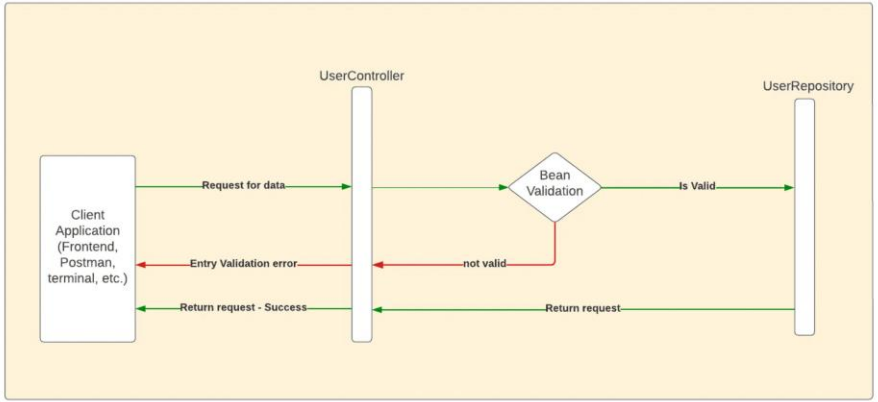
Implementazione Base - Controller

Nel controller REST, l'annotazione **@Valid** sul parametro **@RequestBody** attiva automaticamente la validazione del payload.

Se la validazione fallisce, Spring Boot intercetta l'errore e genera automaticamente un'eccezione **MethodArgumentNotValidException**, interrompendo l'esecuzione del metodo.

```java
@RestController
@RequiredArgsConstructor

public class UtenteController {
    private final UtenteService service;
    @PostMapping("/utenti")
    public ResponseEntity<String> registra(@Valid
                                           @RequestBody Utente utente) {
       service.registra(utente);
       return ResponseEntity.ok("User is valid");
    }
 }
```



Gestione degli Errori

> **La gestione degli errori deve essere esplicita e strutturata.**

Per gestire gli errori in modo coerente, si implementa un metodo annotato con **@ExceptionHandler** che cattura MethodArgumentNotValidException.

Questo metodo estrae i dettagli dal BindingResult e li formatta in una risposta JSON (Mappa campo-errore), permettendo al client di identificare esattamente quali campi non sono validi.

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public Map<String, String> handleValidationExceptions(
    MethodArgumentNotValidException ex) {

    Map<String, String> errors = new HashMap<>();

    ex.getBindingResult().getAllErrors().forEach((error) -> {
        String fieldName = ((FieldError) error).getField();
        String errorMessage = error.getDefaultMessage();
        errors.put(fieldName, errorMessage);
    });

    return errors;
}
```

```java
@Service
@Validated
public class UserService {

    public void createUser(@Valid User user) {
        // logica di creazione...
    }

    public User getUserById(@Min(1) Long id) {
        // logica di recupero...
        return new User();
    }
}
```

Validazione a Livello di Metodo

La validazione non è limitata ai Controller. Applicando l'annotazione **@Validated** a una classe (come un Service), abilitiamo la validazione sui parametri dei suoi metodi.

Quando un metodo viene invocato, Spring intercetta la chiamata e valida i parametri annotati (es. con @Valid o @Min) prima dell'esecuzione.

> **Attenzione: In questo caso viene lanciata una ConstraintViolationException, non MethodArgumentNotValidException.**

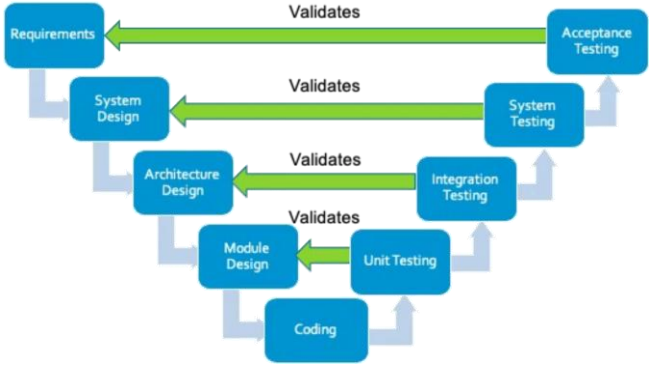
Differenze Tra @Valid e @Validated

@Valid

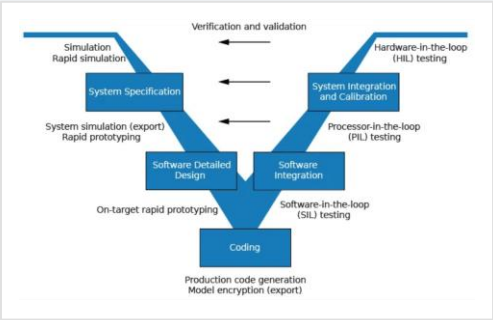
Standard **JSR-303/380**.
Utilizzato principalmente per la **validazione a cascata** (oggetti annidati) e sui parametri dei Controller.

@Validated

Specifica di **Spring Framework**.
Necessaria per abilitare la validazione a **livello di metodo** (es. Service) e per supportare i **Validation Groups**.



Best Practices



01 Messaggi Significativi

Utilizzare sempre messaggi di errore personalizzati che aiutino l'utente a comprendere e correggere l'input.

02 Validazione a Più Livelli

Validare le richieste HTTP nel Controller e la logica di business nei Service per garantire integrità.

03 Gestione Errori Centralizzata

Implementare @ExceptionHandler per restituire risposte JSON strutturate e coerenti.

04 Testing Rigoroso

Scrivere unit test e integration test per verificare che le regole di validazione funzionino come previsto.

05 Semplicità

Usare annotazioni standard quando possibile. Ricorrere ai Custom Validator solo per logiche complesse.

Vantaggi di Spring Validation Starter

Meno Boilerplate

L'approccio dichiarativo riduce drasticamente il codice di validazione manuale (if/else), rendendo il codice più pulito.

Integrazione Nativa

Perfettamente integrato nell'ecosistema Spring Boot: validazione automatica di RequestBody, Entità JPA e parametri.

Standardizzazione

Basato sullo standard JSR-380 (Bean Validation), garantendo portabilità e coerenza con altre librerie Java.

Conclusione

Spring Validation Starter rappresenta lo **standard de facto** per la validazione in Spring Boot. Offre un approccio **dichiarativo ed elegante** che riduce la complessità del codice e migliora la manutenibilità dell'applicazione.

Standard JSR-380

Basato su specifiche solide e collaudate (Hibernate Validator), garantendo portabilità e coerenza.

Integrazione Nativa

Validazione automatica e trasparente su Controller REST, Entità JPA e metodi di Servizio.

Flessibilità Totale

Supporto avanzato per Validation Groups, validatori custom e gestione strutturata degli errori.

GraphQL
Guida Pratica

Dalla teoria all'implementazione con Spring Boot

Scopri come rivoluzionare il recupero dati

Costruisci API moderne ed efficienti

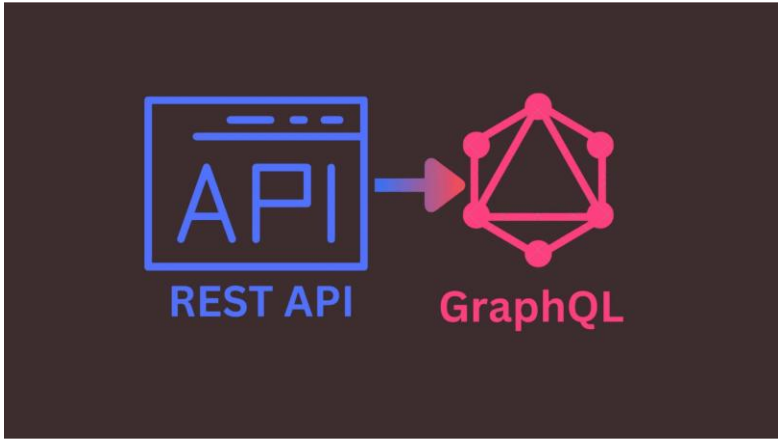
Implementa CRUD con sicurezza integrata

Cos'è GraphQL?

Un Linguaggio di Query Moderno

Un linguaggio di query e runtime per API sviluppato da Facebook. A differenza di REST, **permette ai client di richiedere esattamente i dati di cui hanno bisogno**.

- **Strongly Typed:** Errori rilevati prima dell'esecuzione grazie al sistema di tipi.
- **Single Endpoint:** Un solo URL (/graphql) gestisce tutte le richieste.
- **Strutturato:** La risposta JSON rispecchia esattamente la forma della query.



A Cosa Serve GraphQL?

Risolvere i limiti delle API REST tradizionali

Over-fetching

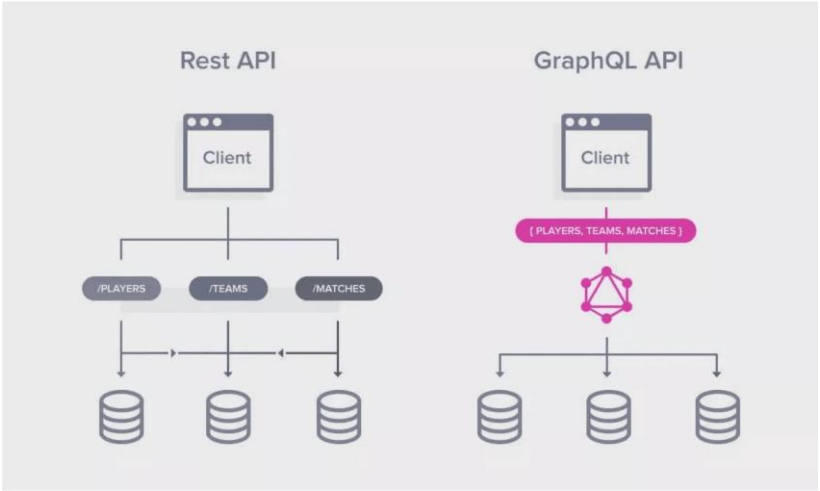
Ricevi troppi dati. Scarichi interi oggetti JSON quando ti serve solo un campo (es. scaricare tutto l'utente solo per il nome).

Under-fetching

Ricevi pochi dati. Sei costretto a fare chiamate multiple (N+1 problem) per ottenere risorse collegate.

La Soluzione

Un'unica richiesta precisa. Ottieni esattamente ciò che chiedi in un solo round-trip.



Vantaggi di GraphQL

Perché è superiore per le applicazioni moderne

Efficienza di Rete

Trasferimento dati minimizzato. Il client scarica solo ciò che serve, ideale per connessioni mobile.

Sviluppo Veloce

Frontend e Backend lavorano in parallelo. Il frontend non deve aspettare nuovi endpoint per ogni modifica UI.

Tipizzazione Forte

Il sistema di tipi previene errori. Se la query è sbagliata, fallisce prima di essere eseguita.

Strumenti Potenti

Introspection permette strumenti incredibili come GraphiQL e Playground per esplorare l'API.

Evoluzione API

Nessun bisogno di versioning (/v1, /v2). Aggiungi nuovi campi senza rompere i client esistenti.

Svantaggi e Limitazioni

Non è una soluzione universale: cosa sapere prima di adottarlo

Curva di Apprendimento

Richiede la comprensione di nuovi concetti come **Schema, Resolver e Type System**. Non è immediato come creare un endpoint REST.

Caching HTTP Complesso

Poiché GraphQL usa quasi sempre POST, non puoi sfruttare il caching nativo del browser o dei CDN come con le GET di REST.

Complessità Iniziale

Il setup richiede più configurazione. Per API molto semplici o microservizi banali, l'overhead di GraphQL potrebbe non valere la pena.

Monitoraggio e Debugging

Gli errori tornano spesso come 200 OK con un payload di errore. Richiede strumenti specifici per tracciare le performance dei singoli campi.

N+1 Problem (Lato Server)

Se non gestito correttamente (es. con DataLoader), il server può eseguire centinaia di query al database per risolvere campi annidati.

Starter e Dipendenze

Il setup essenziale nel file pom.xml

pom.xml

```xml
<!-- Spring Boot GraphQL Starter (Include il supporto nativo) -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-graphql</artifactId>
</dependency>

<!-- Spring Data JPA (Per la persistenza dei dati) -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Il Package GraphQL e lo Schema

Organizzazione del progetto e file essenziali

Struttura Directory

```
src/main/resources/
├── graphql/
│   ├── schema.graphqls # Main schema
│   └── schema/
│       ├── types.graphqls
│       └── queries.graphqls
```

Contenuto Schema

Definizione dei Tipi

Type, Interface, Enum, Scalar. La struttura dati fondamentale.

Query (Lettura)

Le operazioni per recuperare dati. Equivalente alle GET in REST.

Mutation (Scrittura)

Le operazioni per modificare dati (Create, Update, Delete).

Subscription

Operazioni in tempo reale (opzionale) via WebSocket.

Anatomia di uno Schema

Il contratto tra Client e Server

```
type Utente {
  id: ID!
  name: String!
  age: Int
}

type Query {
    getUser(id: ID!): User
    getAllUsers: [User!]!
}

type Mutation {
    createUser(name: String!): User!
}
```

Tipi e Campi

Definiscono la struttura dei dati. `User` è un tipo oggetto con campi specifici.

Non-Null (!)

Il punto esclamativo indica un campo obbligatorio. `String!` non può mai essere null.

Query (Lettura)

Il punto di ingresso per recuperare dati. Equivalente alle GET in REST.

Mutation (Scrittura)

Il punto di ingresso per modificare dati (Create, Update, Delete).

Liste ([ ])

Le parentesi quadre indicano un array. `[User!]!` è una lista non nulla di utenti non nulli.

JPA Model vs GraphQL Schema

Mapping tra Entità Database e Tipi API

Java Entity (DB)

```java
@Entity
public class User {
  @Id @GeneratedValue
  private long id;
  private String name;
  private String email;
  private Integer age;
  @OneToMany(mappedBy = "user")
  private List<Post> posts;
}
```

Map

GraphQL Type (API)

```graphql
type User {
    id: ID!
    name: String!
    email: String!
    age: Int
    # Relazione esposta come array
    posts: [Post!]!
}
```

Il modello JPA definisce come i dati sono salvati. Lo schema GraphQL definisce come i dati sono esposti.

Differenze tra JPA e GraphQL

Cosa cambia tra il Model e lo Schema

Campi Nascosti

Non tutto ciò che è nel DB deve essere nell'API. Password, token interni e flag di sistema rimangono nell'Entity JPA ma non vengono esposti nello Schema.

Campi Calcolati

GraphQL può esporre campi che non esistono nel DB. Esempio: un campo fullName creato concatenando firstName e lastName nel resolver.

Relazioni

In JPA le relazioni sono definite come Lazy/Eager. In GraphQL, è il **client** a decidere se caricare i dati correlati, eliminando l'over-fetching.

Nomi Disaccoppiati

Puoi rinominare i campi nell'API pubblica senza toccare il database. Utile per refactoring o per dare nomi più amichevoli al frontend.

Tipi Custom

Mentre JPA usa tipi Java standard, GraphQL permette di definire Scalar custom (es. Date, Email) per una validazione più stretta.

Implementare una GET (Query)

Dal contratto nello Schema al Resolver Java

1 **Definizione Schema**

```
type Query {
    # Recupera utente per ID
    getUserById(id: ID!): User
    # Recupera tutti gli utenti
    getAllUsers: [User!]!
}
```

2 **Java Resolver (@Controller)**

```
@Controller
@RequiredArgsConstructor

public class UserResolver {
    private final UserRepository repo;
    // Mappa il campo "getUserById" dello schema
    @QueryMapping
    public User getUserById(@Argument Long id) {
      return userRepository.findById(id)
                           .orElse(null);
    }
   // Mappa il campo "getAllUsers"
   @QueryMapping
   public List<User> getAllUsers() {
       return userRepository.findAll(); } }
```

3 **Richiesta Client**

```
query {
    getUserById(id: "1"){
      id
      name
      email
    }
}
```

Implementare una INSERT

Creare nuovi dati con le Mutation

1. Schema Definition

```
input CreateUserInput {
  name: String!
  email: String!
  age: Int
}
type Mutation {
    createUser(input: CreateUserInput!): User!
}
```

2. Java Resolver

```
@MutationMapping
public User createUser(
        @Argument CreateUserInput input) {
    User user = new User();
    user.setName(input.getName());
    user.setEmail(input.getEmail());
    return userRepository.save(user);
}
```

3. Client Request

```
mutation{createUser(input: {name:"Mario",email:"mario@example.com",age:30}) {idname}}
```

Implementare un UPDATE

Modifica parziale dei dati esistenti

Schema Definition

schema.graphqls

```
input UpdateUserInput {
  id: ID!
  # Campi opzionali per update parziale
  name: String
  email: String
  age: Int
}
type Mutation { updateUser(input: UpdateUserInput!): User }
```

Client Request

Query

```
mutation { updateUser(input: { id: "1", name: "Luigi" }) { idnameemail# Restituisce
il valore aggiornato } }
```

Java Resolver Logic

Controller.java

```
@MutationMapping
public User updateUser(@Argument UpdateUserInput input) {
  // 1. Recupera l'entità esistente
  User user = repo.findById(input.getId())
                  .orElseThrow(() ->
        new RuntimeException("Non trovato"));
  // 2. Aggiorna SOLO se il campo è presente (non null)
  if (input.getName() != null)
    user.setName(input.getName());
  if (input.getEmail() != null)
    user.setEmail(input.getEmail());
  if (input.getAge() != null)
    user.setAge(input.getAge());
  // 3. Salva e restituisce
  return repo.save(user);
}
```

Implementare una DELETE

Rimuovere dati e confermare l'operazione

Schema Definition

```
typeMutation { # Restituisce true se eliminatodeleteUser(id: ID!):
Boolean! }
```

Client Request

```
mutation { deleteUser(id: "1") }
```

Best Practice: È utile restituire un booleano per confermare il successo, oppure l'ID dell'oggetto eliminato per aggiornare la cache del client.

Java Resolver Logic

```java
@MutationMapping
public Boolean deleteUser(@Argument long id) {
   // 1. Verifica esistenza
    if (!repo.existsById(id))
      throw new RuntimeException("User not found");
  // 2. Esegui eliminazione
   repo.deleteById(id);
  // 3. Conferma successo
   return true;
}
```

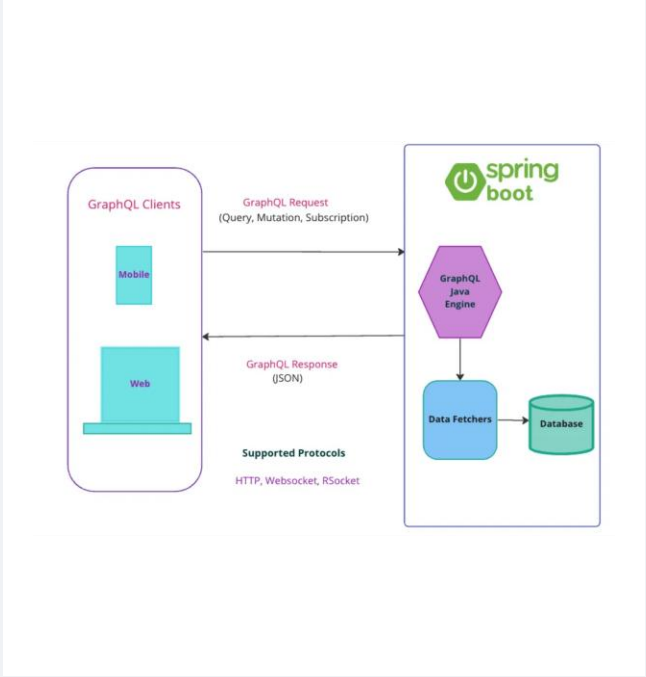
Sicurezza: Gestione degli Endpoint

Proteggere l'unico punto di ingresso

Single Endpoint Challenge

A differenza di REST, GraphQL espone un solo URL (/graphql). Non possiamo usare regole basate sul path (es. /admin/**) per proteggere risorse specifiche. La sicurezza si sposta dal livello HTTP al livello applicativo.

```
@Configuration@EnableWebSecuritypublic classSecurityConfig {
@BeanpublicSecurityFilterChainfilterChain(HttpSecurity http) { return http .csrf(csrf ->
csrf.disable()) .authorizeHttpRequests(auth -> auth // Protegge l'accesso all'endpoint
.requestMatchers("/graphql").authenticated() .anyRequest().permitAll() ) .build(); } }
```



OncePerRequestFilter

Intercettare la richiesta prima di GraphQL

Esecuzione Garantita

Assicura che il filtro venga eseguito una sola volta per richiesta HTTP, evitando duplicazioni in caso di forward interni.

Context Setup

È il luogo ideale per validare il token JWT e popolare il SecurityContextHolder.

Pre-GraphQL

Avviene prima che l'engine GraphQL processi la query, proteggendo l'intero endpoint.

JwtAuthenticationFilter.java

```
@Component public class JwtAuthFilter extends OncePerRequestFilter { @Override
protected void doFilterInternal (HttpServletRequest req , HttpServletResponse res ,
FilterChain chain ) { String jwt = extractJwtFromRequest (req ); if ( jwt != null &&
tokenProvider .validateToken (jwt )) { // 1. Estrae dettagli utente dal token UserDetails user =
tokenProvider .getUser (jwt ); // 2. Crea oggetto Authentication var auth = new
UsernamePasswordAuthenticationToken ( user , null , user .getAuthorities ());
// 3. Imposta il Context (CRUCIALE)
SecurityContextHolder.getContext().setAuthentication(auth);
} chain .doFilter (req , res ); } }
```

Autorizzazione nei Resolver

Controllo granulare degli accessi con @PreAuthorize

UserResolver.java

```java
@Component
@EnableMethodSecurity
public class UserResolver {
  @QueryMapping
  @PreAuthorize("hasRole('USER')")
  public User getUserById(@Argument Long id) {
      return repo.findById(id).orElse(null);
  }
  @MutationMapping
  @PreAuthorize("hasRole('ADMIN')")
  public User createUser(@Argument Input in) {
    // Solo admin possono creare utenti
    User user = new User(in);
    return repo.save(user);
  }
}
```

Controllo Granulare

A differenza di REST dove proteggi l'URL, in GraphQL proteggi il **metodo**. Puoi definire regole diverse per ogni campo o operazione.

Gestione Ruoli

Usa le annotazioni standard di Spring Security.

ROLE_USER	Lettura dati
ROLE_ADMIN	Modifica dati

Testabilità

La logica di sicurezza è dichiarativa e vicina al codice di business, rendendo facile verificare i permessi durante i test unitari.

Gestione degli Errori

Eccezioni Java e Risposte Strutturate

1. Eccezione Custom

```
public class GraphQLException extends RuntimeException {
   private String code;
   public GraphQLException(String msg, String code) {
     super(msg);
     this.code = code;
} }
```

2. Lancio nel Resolver

```
@QueryMapping
public User getUserById(@Argument Long id) {
return repo.findById(id).orElseThrow(() ->
     new GraphQLException( "User " + id + " not found", "USER_NOT_FOUND" ));
}
```

3. Risposta Client (JSON)

```
{
  "data": null,
"errors": [
{
"message": "User 999 not found", "locations": [ ... ],
"path": ["getUserById"],
"extensions": {
"code": "USER_NOT_FOUND",
"classification": "DataFetchingException"
} } ] }
```

Nota: GraphQL restituisce spesso HTTP 200 anche in caso di errori applicativi. Il client deve controllare se il campo errors è presente.

Gestione degli Errori

Implementazione del Resolver

UserResolver.java

```java
@Component
public class GraphQLExceptionResolver extends DataFetcherExceptionResolverAdapter {

    @Override
    protected GraphQLError resolveToSingleError(Throwable ex, DataFetchingEnvironment env) {


        if(ex instanceof GraphQLException){
            return GraphqlErrorBuilder.newError(env)
                    .message(ex.getMessage())
                    .errorType(ErrorType.BAD_REQUEST)
                    .build();
        }

        return GraphqlErrorBuilder.newError(env)
                .message("Errore interno")
                .errorType(ErrorType.INTERNAL_ERROR)
                .build();
    }
}
```

In GraphQL la convenzione è diversa, non abbiamo la possibilità di lavorare con un @RestControllerAdvice e di utilizzare L'@ExceptionHandler

Quello che andremo ad utilizzare sarà un Resolver centralizzato che gestirà tutti i possibili problemi, questo Resolver sarà un @Component e lo otterremo estendendo DataFetcherExceptionResolverAdapter

I vantaggi sono:

- Gestione Centralizzata dei problem
- Funziona sia per Query che per Mutation
- Mappa correttamente tutti gli errori

Ottimizzazione Performance

Risolvere il problema N+1 con DataLoader

⚠ Il Problema N+1

Senza ottimizzazione, GraphQL esegue una query per ogni campo risolto. Se chiedi 10 utenti e i loro post, farai **1 query per gli utenti + 10 query per i post**.

```
SELECT * FROM users
  SELECT * FROM posts WHERE user_id = 1
  SELECT * FROM posts WHERE user_id = 2
  ... (ripetuto N volte)
  SELECT * FROM posts WHERE user_id = 10
```

La Soluzione: Batching

DataLoader "aspetta" brevemente, raccoglie tutti gli ID richiesti e li risolve con **un'unica query ottimizzata**.

```
SELECT * FROM users
SELECT * FROM posts WHERE user_id IN (1, 2, ..., 10)
```

```
BatchLoader<Long, List<Post>> postLoader = ids -> { return
postRepository.findByUserIdIn(ids); };
```

Implementare DataLoader

Uso di @BatchMapping in Spring Boot

Dove si mette?

Il metodo va nel **Controller**. È responsabile di risolvere un campo dello schema, ma delega il recupero dati al Repository.

Come funziona?

Spring raccoglie automaticamente gli oggetti padre (es. List<Author>) e chiama il metodo una sola volta.

Devi restituire una **Map** che collega ogni Autore alla sua lista di Post.

AuthorController.java

```java
@Controller public class AuthorController {
  // Sostituisce @SchemaMapping per performance
    @BatchMapping
    public Map<Author, List<Post>> posts(List<Author> authors) {
        // 1. Estrai tutti gli ID
        List<Long> authorIds = authors.stream().map(Author::id).toList();
        // 2. Esegui UNA sola query (WHERE IN)
        List<Post> allPosts = postRepo.findByAuthorIdIn(authorIds);
        // 3. Raggruppa i risultati per Autore
        return allPosts.stream()
                .collect(Collectors.groupingBy(post ->authors.stream().filter(a ->a.id().equals(post.authorId())))
                .findFirst()
                .get()));
    }
}
```

Esempio Pratico: Blog Engine

Modellare le relazioni tra entità

Author

L'utente che scrive i contenuti. Ha una lista di Post.

↓ 1:N

Post

L'articolo del blog. Appartiene a un Author e ha molti Commenti.

↓ 1:N

Comment

Feedback dei lettori. Collegato a un singolo Post.

schema.graphqls

```
type Author {
  id: ID!
  name: String!
  posts: [Post]!
}
type Post {
  id: ID!
  title: String!
  content: String
  author: Author!
  comments: [Comment]!
}
type Comment {
  id: ID!
  text: String!
}
```

Esempio Pratico: Implementazione Java

Mapping delle entità e risoluzione delle relazioni

1. Data Records

```
public record Author(
        Long id,
        String name) {}
public record Post(
        Long id,
        String title,
        Long authorId ) {}
```

Nota: I record Java mappano automaticamente i campi dello Schema GraphQL se i nomi coincidono.

2. BlogController

```
@Controller
public class BlogController {
  @QueryMapping
  public Author author(@Argument Long id) {
  return authorRepo.findById(id);
  }
  // Risolve il campo "posts" dell'Author
  @SchemaMapping
  public List<Post> posts(Author author) {
    return postRepo.findByAuthorId(author.id());
  }
}
```

Esempio Pratico: Query Complessa

Recuperare dati annidati in una singola chiamata

Client Request

```
query {
  author(id: "1") {
    name
    posts {
      title
      comments {
        text
      }
    }
  }
}
```

Server Response

```
{
  "data":{
    "author":{
      "name": "Mario Rossi",
      "posts": [ {
        "title": "Intro a GraphQL",
        "comments": [ {
          "text": "Ottimo articolo!"
        }, {
          "text": "Molto chiaro."
        } ]
      } ]
    }
  }
}
```

Best Practices

Consigli per uno sviluppo robusto e scalabile

Naming Convention

Usa nomi descrittivi e coerenti tra Schema e Resolver.

Validazione Input

Valida sempre gli input nel Resolver e nel database.

Paginazione

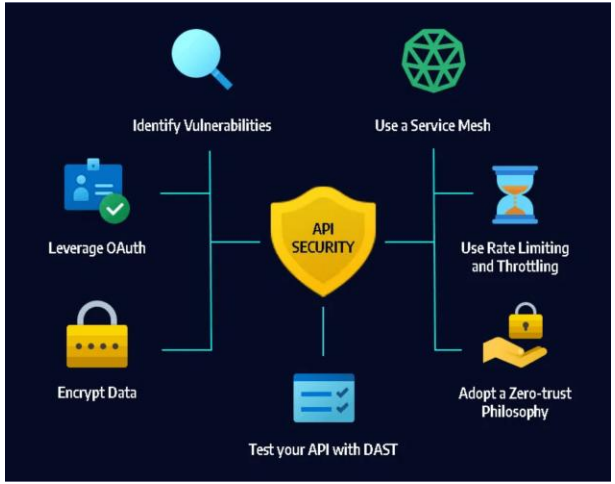
Implementa paginazione cursor-based per dataset grandi.

Testing Automatizzato

Testa query e mutation con integrazione (GraphQlTester / MockMvc).

Performance & Caching

Usa DataLoader per N+1 e abilita caching dove utile.



Conclusione

GraphQL: Il Futuro delle API Moderne

Punti Chiave

- **Efficienza:** Un solo endpoint, zero over-fetching, dati precisi.
- **Affidabilità:** Schema fortemente tipizzato che previene errori.
- **Sicurezza:** Controllo granulare degli accessi a livello di resolver.

Prossimi Passi

Pratica con piccoli progetti Spring Boot

Esplora GraphQL Playground

Studia pattern avanzati come DataLoader