



Spring Boot

Risposta HTTP





Risposta HTTP

- Fornire al client risposte HTTP che contengono:
 - Codice di stato
 - Corpo della risposta
 - Headers aggiuntivi(opzionale)
- Per raggiungere questo obiettivo ci sono diverse modalità.
 - ResponseEntity<>
 - Altre..





ResponseEntity<T>

```
@PostMapping("/hello/{nome}")
public ResponseEntity<String> hello(@PathVariable("nome") String nome){
    return new ResponseEntity<>("ciao "+nome, HttpStatus.OK);
}
```

```
@PostMapping("/hello")
public ResponseEntity<String> hello(@RequestParam("nome") String nome){
    if(nome==null || nome.isBlank()) return ResponseEntity.badRequest().build();
    return ResponseEntity.status(HttpStatus.OK).body("Ciao "+nome);
}
```





Spring Boot

Pattern DTO





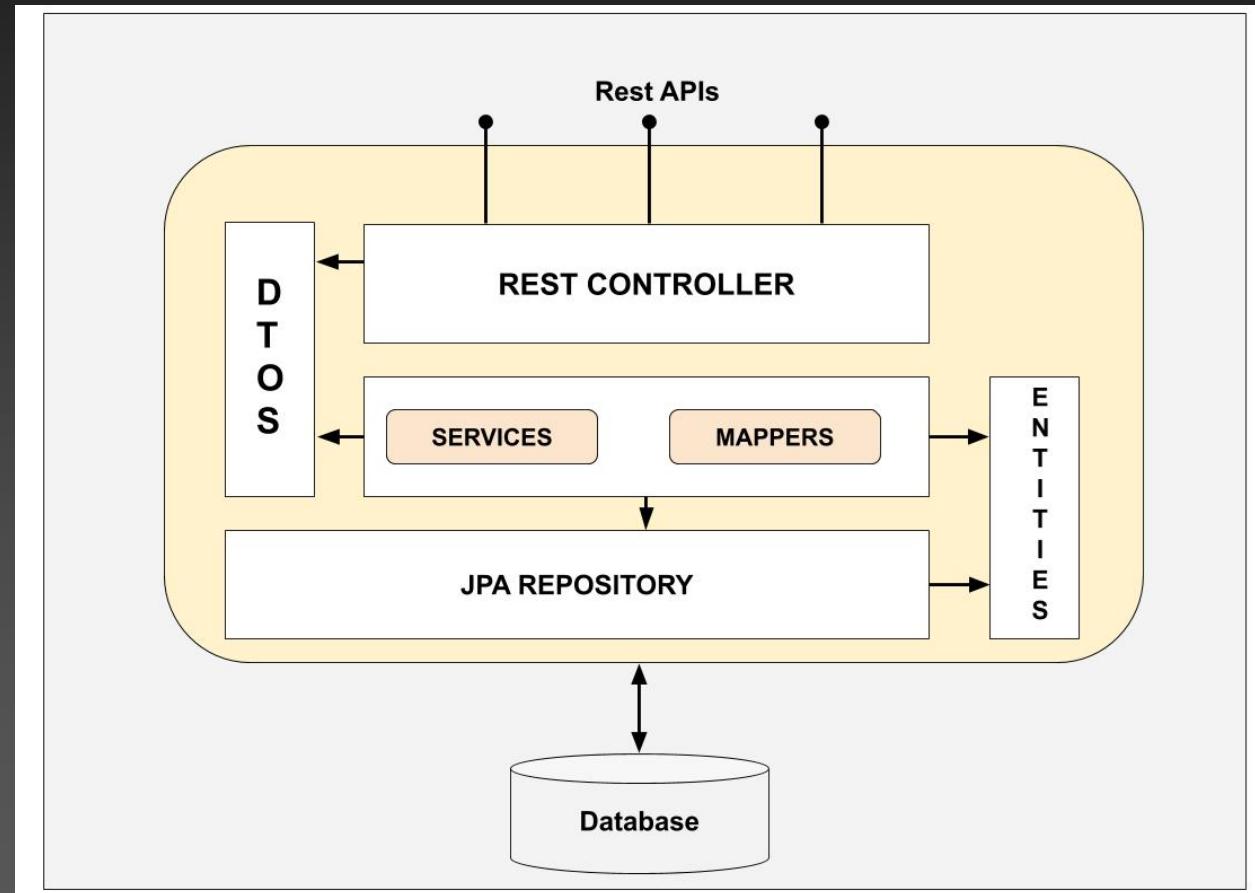
Pattern DTO

- I DTO o Data Transfer Object sono degli oggetti POJO che encapsulano dati e non contengono logica aziendale.
- Tali oggetti vengono inviati tra i componenti dell'applicazione.
- Lo scopo è di ridurre il numero di chiamate tra componenti.(es: tra client e server, server e server). Raggruppando più parametri all'interno di una singola chiamata.
- Vantaggi:
 - Disaccoppiamento delle classi entity dal layer controller.
 - Risposta ottimizzata in base alle esigenze del client.
 - Sicurezza.
 - Non più metodi che prendono tanti parametri in input.
- Svantaggi:
 - La conversione da entity a DTO e viceversa può essere «costoso».



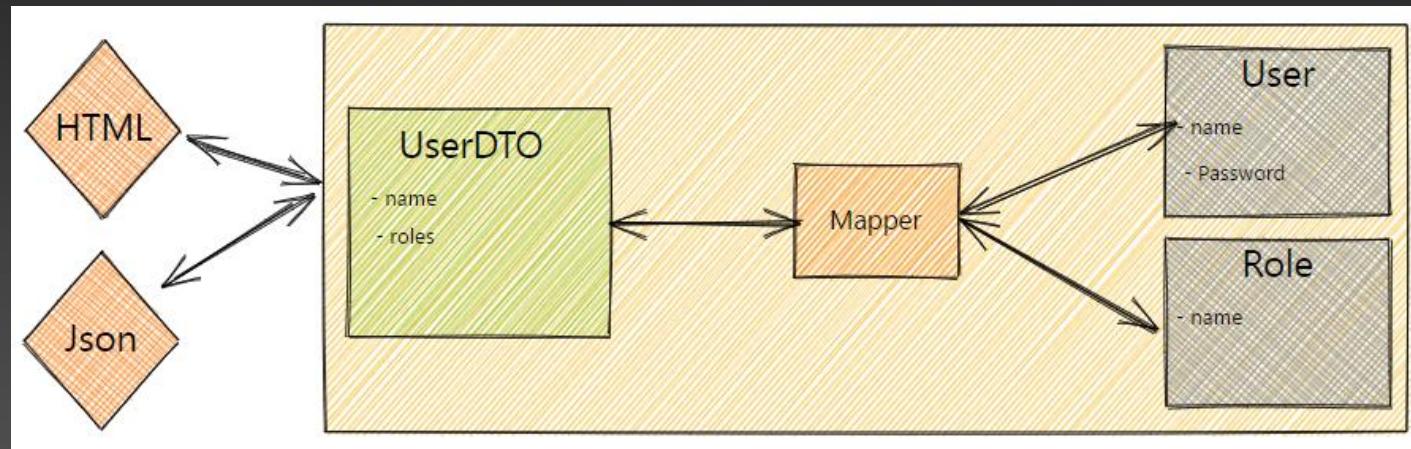


Pattern DTO





Pattern DTO





Pattern DTO

```
@Entity  
public class Utente {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private long id;  
    private String email;  
    private String password;  
  
    @ManyToMany  
    @JoinTable(name = "ruolo_utente",  
              joinColumns = @JoinColumn(name = "id_utente", nullable = false),  
              inverseJoinColumns = @JoinColumn(name = "id_ruolo", nullable = false))  
    private List<Ruolo> ruoli;  
  
}
```

```
@Entity  
public class Ruolo {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private long id;  
    private String nome;  
  
}
```

```
public class LoginDTO {  
    private String username;  
    private String password;  
  
    //getter - setter - costruttore  
  
}
```

```
public class UtenteDTO {  
  
    private String nome;  
    private List<String> ruoli;  
  
    //getter - setter - costruttori  
  
}
```





Pattern DTO

```
@Component
public class UtenteMapper {

    public UtenteDTO toDTO(Utente u){
        String nome=u.getNome();
        List<String> ruoli=
        u.getRuoli().stream().map(Ruolo::getNome).toList();
        return new UtenteDTO(nome,ruoli);
    }
}
```





Pattern DTO

```
@PostMapping("/utente/login")
public ResponseEntity<UtenteDTO> login(@RequestBody LoginDTO request){
    return ResponseEntity.status(HttpStatus.OK).body(service.login(request.getUsername(),request.getPassword()));
}
```

```
@Service
public class UtenteServiceImpl implements UtenteService{

    private final UtenteRepository repo;
    private final UtenteMapper mapper;

    public UtenteServiceImpl(UtenteRepository repo, UtenteMapper mapper) {
        this.repo = repo;
        this.mapper = mapper;
    }

    @Override
    public UtenteDTO login(String email, String password){
        Utente u=repo.findByEmailAndPassword(email,password).orElseThrow(UtenteNonTrovatoException::new);
        return mapper.toDTO(u);
    }
}
```





Pattern DTO

HTTP request body

```
1 {  
2   ... "username": "admin@email.com",  
3   ... "password": "root"  
4 }  
5
```

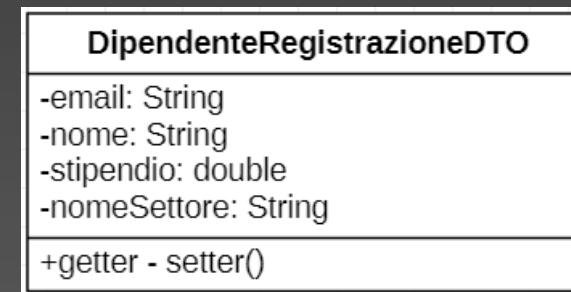
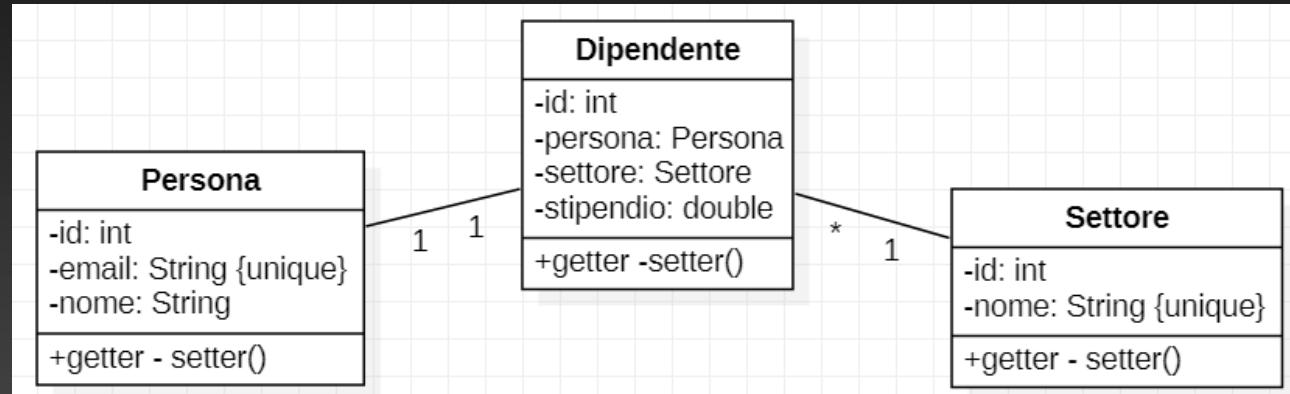
HTTP response body

```
1 [  
2   {  
3     "nome": "admin",  
4     "ruoli": [  
5       "admin"  
6     ]  
7   ]  
8
```





Pattern DTO





Spring Boot

Gestione Errori





Gestione errori

- Prima di Spring 3.2:
 - HandlerExceptionResolver interface
 - @ExceptionHandler
- Dalla versione di Spring 3.2 abbiamo @RestControllerAdvice
- Dalla versione di Spring 5 abbiamo la classe ResponseStatusException
- In alcuni contesti conviene usare un mix tra le varie modalità





@RestControllerAdvice

- In passato, la gestione delle eccezioni generate dagli endpoint di un controller avvenivano dentro un particolare metodo della classe annotato con @ExceptionHandler

```
@RestControllerAdvice  
public class GestoreErrori {  
  
    @ExceptionHandler({MiaException1.class, SQLIntegrityConstraintViolationException.class})  
    public ResponseEntity<MessaggioErrore> handleException(Exception e){  
        //corpoDelMetodo  
        System.out.println(e.getMessage());  
        return null;  
    }  
}
```

- Questo sistema ha un grande svantaggio:
- Un metodo che gestisce le eccezioni per ogni controller





@RestControllerAdvice

- Con l'annotazione `@RestControllerAdvice` riusciamo ad avere una gestione delle eccezioni globale all'interno dell'applicazione.
- Non più metodi annotati `@ExceptionHandler` sparsi per i controller dell'applicazione.
- Vantaggi:
 - Pieno controllo sul corpo della risposta e sul codice si stato.
 - Mappatura di diverse eccezioni allo stesso metodo, da gestire insieme.
 - Utilizzo della moderna `ResponseEntity<T>`.





@RestControllerAdvice

```
{  
    "id":0,  
    "nome": null,  
    "email":"utente@gmail.com",  
    "eta": 18  
}
```

```
@RestController  
public class UtenteController {  
  
    @Autowired  
    UtenteService service;  
  
    @PostMapping("/utente/add")  
    public ResponseEntity<Void> registrazione(@RequestBody Utente u) throws DatiNonValidiException{  
        service.save(u);  
        return ResponseEntity.status(HttpStatus.CREATED).build();  
    }  
}
```





@RestControllerAdvice

```
@Service
public class UtenteServiceImpl implements UtenteService{
    @Autowired
    UtenteRepository repo;
    @Override
    public void save(Utente u) throws DatiNonValidiException {
        if(u.getNome()==null||u.getNome().isBlank()||
           u.getEmail()==null||u.getEmail().isBlank()||
           u.getEta()<18||u.getEta()>100) throw new DatiNonValidiException("valori utente null,vuoti o non validi");
        repo.save(u);
    }
}
```

```
public class DatiNonValidiException extends Exception {

    public DatiNonValidiException(String msg){
        super(msg);
    }
}
```





@RestControllerAdvice

```
@RestControllerAdvice  
public class GestoreErrori {  
  
    @ExceptionHandler({DatiNonValidiException.class})  
    public ResponseEntity<MessaggioErrore> gestoreDatiNonValidi(DatiNonValidiException e, WebRequest wr){  
        MessaggioErrore m=new MessaggioErrore(HttpStatus.BAD_REQUEST.value(),  
                                         HttpStatus.BAD_REQUEST.name(),  
                                         e.getMessage(),  
                                         wr.getDescription(false));  
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(m);  
    }  
}
```

```
public class MessaggioErrore {  
    private LocalDateTime timestamp;  
    private int status;  
    private String error, message, path;  
}
```





@RestControllerAdvice

Body Cookies Headers (4) Test Results

400 Bad Request 69 ms 317 B Save Response ▾

Pretty Raw Preview Visualize JSON

```
1  {
2      "timestamp": "2022-10-04T14:54:11.766+00:00",
3      "status": 400,
4      "error": "BAD_REQUEST",
5      "message": "valori utente null, vuoti o non validi",
6      "path": "uri=/api/v1/utente/registrazione"
7 }
```





ResponseStatusException

- Vantaggi:
 - Eccellente per la prototipazione: implementabile velocemente.
 - Un tipo di eccezione può essere abbinato a codice di stato diversi.
 - Non dobbiamo creare tante classi di eccezioni custom.
 - Maggiore controllo sulla gestione delle eccezioni, la gestione è all'interno del metodo.





ResponseStatusException

```
@RestController
public class UtenteController {

    @Autowired
    UtenteServiceImpl service;

    @PostMapping("/utente/add")
    public ResponseEntity<Void> registrazione(@RequestBody Utente u) throws DatiNonValidiException{
        try {
            service.save(u);
            return ResponseEntity.status(HttpStatus.CREATED).build();
        }catch (DatiNonValidiException ex){
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST,"utente già presente o dati immessi non validi");
        }
    }
}
```

GET http://localhost:8080/api/v1/utente/1001

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

1 "timestamp": "2022-10-03T18:12:11.509+00:00",
2 "status": 404,
3 "error": "Not Found",
4 "message": "Utente non trovato",
5 "path": "/api/v1/utente/1001"





ResponseStatusException

- Le informazioni contenute nella response variano in base alla configurazione stabilita nell' application.properties o yml.
- Ad esempio:

```
server.error.include-stacktrace=always
```

- Tramite questa configurazione al client NON verrà inviato il trace dell'eccezione.





Gestione delle eccezioni

- Entrambe le soluzioni possono convivere:
- Alcune eccezioni gestite con ResponseStatusException e altre con @RestControllerAdvice. NON gestire lo stesso tipo di eccezione con entrambi i modi.





Spring Boot

Spring Data JPA





Tipi di Query

- All'interno delle nostre repository possiamo definire 3 tipi di query
- Query native
 - Sono contraddistinte dal parametro native = true e sono scritte in sql puro, si interfacciano con i campi del db
- Query in JPQL
 - Sono scritte in JPQL e si interfacciano con i campi delle entity Java
- Query derivate
 - Sono solo definite, devono combaciare con i campi delle entity Java





Repository layer

```
public interface OrdineRepository extends JpaRepository<Ordine,Long> {  
  
    @Query(nativeQuery = true, value = "select * from Ordine where data_inizio>= :dataInizio and data_fine  
    <= :dataFine")  
    List<Ordine> trovaPerData(LocalDate dataInizio, LocalDate dataFine);  
  
    @Query("select o from Ordine o where o.id = :id")  
    Optional<Ordine> trovaPerId(Long id);  
  
    List<Ordine> findAllByCliente_IdAndDisattivatoIsFalse(Long id);  
  
}
```





Spring

JWT



JWT



- Json Web Token è un sistema di autenticazione basato su un oggetto di tipo Json.
- Le informazioni vengono memorizzate in una stringa alfanumerica chiamata Token.
- Il token è suddiviso in tre parti.
 - Header – contiene le informazioni di codifica del token
 - PayLoad – contiene i nostri dati
 - Signature – ci permette di controllare se il token è stato modificato
- Possiamo utilizzare JWT per controllare se:
 - Un utente utilizza delle credenziali esatte – autenticazione
 - Un utente ha accesso a delle specifiche risorse – autorizzazione
 - Da quanto tempo l'utente ha effettuato una determinata operazione – Validazione



JWT



- Quello che andremo a fare è creare un server esterno di validazione che ci permetterà di creare il nostro token
- Nella nuova applicazione andiamo ad importare lo starter security e jjwt che ci servirà per creare effettivamente il nostro server

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.13.0</version>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.13.0</version>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId>
  <version>0.13.0</version>
</dependency>
```

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



JWT



- Per gestire JWT dobbiamo creare un RestController con tre metodi
 - Il primo per creare un Token
 - Il secondo per refreshare il token
 - Il terzo sarà l'ExceptionHandler





- Dovremmo anche modificare l'utente facendo implementare UserDetails
- Partiamo da un esempio con due ruoli, un utente e un gestore

```
public enum Ruolo {  
    UTENTE("UTENTE"),  
    GESTORE("GESTORE");  
    private String ruolo;  
    Ruolo(String ruolo){  
        this.ruolo=ruolo;  
    }  
    public String getNome(){return ruolo;}  
}
```

JWT



```
public class Utente implements UserDetails {  
  
    private Ruolo ruolo;  
    private String username;  
    private String password;  
  
    //costruttori e metodi della classe  
  
}
```





- Tra i vari metodi che dovremmo implementare sull'utente ci sarà quello che ci permetterà di vedere i ruoli di quell'utente

```
@Override  
public Collection<? extends GrantedAuthority> getAuthorities() {  
    return List.of(new SimpleGrantedAuthority("ROLE_"+ruolo.getNome()));  
}
```



- Oltre a questo avremo bisogno di:
 - Un service per la gestione del token
 - Una classe di configurazione con i bean di configurazione
 - Un filter di autenticazione ottenuto estendendo «OncePerRequestFilter»
 - Una classe di configurazione che ci permetterà di gestire la filterChain
 - La classe che gestirà i Ruoli che hanno accesso (l'utente) dovrà estendere «UserDetails»



JWT – service gestione token

- Il serice per la generazione e gestione dei token potremmo idealmente dividerlo in 2 parti, la prima per la generazione, la seconda per la validazione
- Entrambe utilizzeranno una javax.crypto.SecretKey di sicurezza che servirà per validare il token

```
@Service
public class GestoreToken {

    private static final String SECRET_KEY = "chiaveAbbastanzaLunga";

    private SecretKey getSignInKey() {
        return Keys.hmacShaKeyFor(Decoders.BASE64.decode(SECRET_KEY.getBytes()));
    }

}
```

JWT – service gestione token - claims



- All'interno del nostro token potremo avere quante informazioni vogliamo, tutte queste informazioni faranno parte di un oggetto chiamato Claims che sarà contenuto nel nostro token
- Il token che genereremo sarà inserito nell'header, oltre ai claims possiamo firmarlo con la nostra key, e inserire tutte le info che vogliamo



JWT – service gestione token

```
public String generaToken(Utente u){  
    String ruolo=u.getRuolo().toString();  
    String username=u.getEmail();  
    String dataNascita=u.getDataDiNascita().format(DateTimeFormatter.ofPattern("EEEE dd MMMM yyyy"));  
    String saluto="ciao, hai letto i miei dati, e adesso?";  
  
    //1000L (1sec)*60(1 min)*60(1h)*24(1g)*60(60g)  
    long millisecondiDiDurata=1000L*60*60*24*60;  
    String token=Jwt.tokens.builder()  
        .claims()  
        .add("ruolo",ruolo)  
        .add("dataNascita",dataNascita)  
        .add("saluto",saluto)  
            //il subject di solito viene utilizzato per la convalida, ma come tutti è un tag opzionale  
        .subject(username)  
            //l'issuedAt è la data di creazione del token  
        .issuedAt(new Date(System.currentTimeMillis()))  
            //l'expiration è la data di scadenza del token  
        .expiration(new Date(System.currentTimeMillis()+millisecondiDiDurata))  
        .and()  
            //la parte del signWith ci permette di "firmare" il nostro token e renderlo non modificabile  
        .signWith(getSignInKey())  
    .compact();  
    return token;  
}
```



JWT – service gestione token - claims



- Tutti i dati inseriti saranno parte dei Claims, che posso ripendere dal token, se la firma del token è sbagliata o il token è scaduto verrò lanciata un'exception

```
private Claims estraiClaims(String token) {  
    return (Claims)Jwts  
        .parser()  
        .verifyWith(getSignInKey())  
        .build()  
        .parse(token)  
        .getPayload();  
}
```



JWT – service gestione token - claims



- Una volta ottenuti i claims potremo prendere le info contenute come username e data di scadenza del token

```
private boolean isTokenScaduto(String token) {  
    return getDataScadenza(token).before(new Date());  
}  
  
private Date getDataScadenza(String token) {  
    return estraiClaims(token).getExpiration();  
}  
  
public String getUsername(String token) {  
    return estraiClaims(token).getSubject();  
}
```





JWT – service gestione token

- Ora finalmente potremmo sapere se il token è ancora valido controllando tutti i parametri dell'utente presi dal database

```
public boolean isTokenValid(String token) {  
    String email=getUsername(token)  
    Utente u=service.findByEmail(email);  
    boolean b1= u.isUtenteAbilitato();  
    boolean b2= !isTokenScaduto(token);  
    boolean b3= !u.isScaduto();  
    return b1&&b2&&b3;  
}
```



JWT – bean di configurazione



- Altra parte fondamentale è una classe di configurazione annotata come `@Configuration` che ci permetterà di creare dei bean per configurare tutti gli oggetti di cui avremo bisogno per la nostra security, questa classe utilizzerà un oggetto per il `findByEmail` o `findByUsername`

```
@Configuration  
@RequiredArgsConstructor  
public class ConfigurationBean {  
  
    private final UtenteService service;  
  
}
```





JWT – bean di configurazione

- I bean contenuti in questa classe saranno 4, il primo si baserà sulla repository dell'utente e ci permetterà, partendo dallo username di prendere l'utente

```
@Bean  
protected UserDetailsService userDetailsService() {  
    return username -> service.findByEmail(username)  
        .orElseThrow(  
    () -> new ResponseStatusException(HttpStatus.NOT_FOUND,"nessun utente nel sistema con l'email selezionata")  
        );  
}
```

- Il secondo servirà per creare un PasswordEncoder,

```
@Bean  
protected PasswordEncoder passwordEncoder() {  
    return new BCryptPasswordEncoder();  
}
```





JWT – bean di configurazione

- Il terzo prenderà il manager di autenticazione dalle configurazioni di default

```
@Bean  
protected AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws  
Exception {  
    return config.getAuthenticationManager();  
}
```

- L'ultimo gestirà il provider di autenticazione utilizzando gli altri bean

```
@Bean  
protected AuthenticationProvider authenticationProvider() {  
    DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider(userDetailsService());  
    authProvider.setPasswordEncoder(passwordEncoder());  
    return authProvider;  
}
```





JWT – OncePerRequestFilter

- Il filter di autenticazione sarà il punto centrale della nostra implementazione
- In questa parte del codice controlleremo che:
 - il token non sia scaduto
 - che l'utente abbia realmente i permessi per accedere alla risorsa richiesta
- Qui avremo bisogno del service di gestione dei token e del service dello UserDetails
- Il metodo più importante è il doFilterInternal che farà l'intero controllo



JWT – OncePerRequestFilter



Override

```
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws  
ServletException, IOException {  
    String token=request.getHeader("Authorization");  
  
    if(token!=null && token.startsWith("Bearer")) {  
        Utente user=gestoreToken.findByToken(token);  
        if (SecurityContextHolder.getContext().getAuthentication() == null) {  
            UsernamePasswordAuthenticationToken upat = new UsernamePasswordAuthenticationToken(user, null,  
user.getAuthorities());  
            upat.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));  
            SecurityContextHolder.getContext().setAuthentication(upat);  
        }  
        chain.doFilter(request,response);  
    }  
}
```





JWT – Configuratore Path

- L'ultima classe da creare sarà la classe di configuazione che conterrà il bean per la creazione della SecurityFilterChain
- Qui avremo bisogno del filter di autenticazione (OncePerRequestFilter) e dell'AuthenticationProvider
- Andremo a disabilitare le csrf e la session e abiliteremo i path che ci interessano solo agli utenti che hanno il permesso per quei path
- Oltre all'annotation **@Configuration** la classe avrà anche l'annotation **@EnableWebSecurity**





JWT – SecurityFilterChain da 3.1

- ```
@Configuration
@EnableWebSecurity

public class ConfiguratorePath {

 private final FilterAutenticazione jwtAuthFilter;
 private final AuthenticationProvider authenticationProvider;

 public ConfiguratorePath(FilterAutenticazione jwtAuthFilter, AuthenticationProvider authenticationProvider) {
 this.jwtAuthFilter = jwtAuthFilter;
 this.authenticationProvider = authenticationProvider;
 }

 @Bean
 protected SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
 http
 .csrf(AbstractHttpConfigurer::disable)
 .authorizeHttpRequests(
 auth -> auth
 .requestMatchers("/all/**").permitAll()
 .requestMatchers("/admin/**").hasRole(GESTORE.getNome())
 .requestMatchers("/studente/**").hasAnyRole(STUDENTE.getNome(); GESTORE.getNome())
 .anyRequest().authenticated()
)
 .sessionManagement(sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
 .cors(AbstractHttpConfigurer::disable)
 .authenticationProvider(authenticationProvider)
 .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);
 }
 return http.build();
 }
}
```





# JWT – Gestire la login

- Nella login non torneremo più l'utente ma inseriremo come header il nostro token con la chiave «Authorization»

```
@PostMapping("/login")
public ResponseEntity<String> login(@RequestBody LoginRequest request) {
 Utente u=utenteService.login(request);
 return ResponseEntity.status(HttpStatus.OK).header("Authorization",
gestoreToken.generaToken(u)).build();
}
```





# JWT – Prendere l'utente nel metodo

- Per prendere l'utente negli hand point possiamo utilizzare come parametro d'ingresso un oggetto di tipo UsernamePasswordAuthenticationToken
- Da questo oggetto potremmo prendere il principal
- Il metodo getPrincipal() tornerà un Object che sarà il nostro utente

```
@GetMapping("/getUtente")
public ResponseEntity<String> prendiUtente(@AuthenticationPrincipal UserDetails token){
 Utente u=(Utente)token;
 return u.getEmail();
}
```

