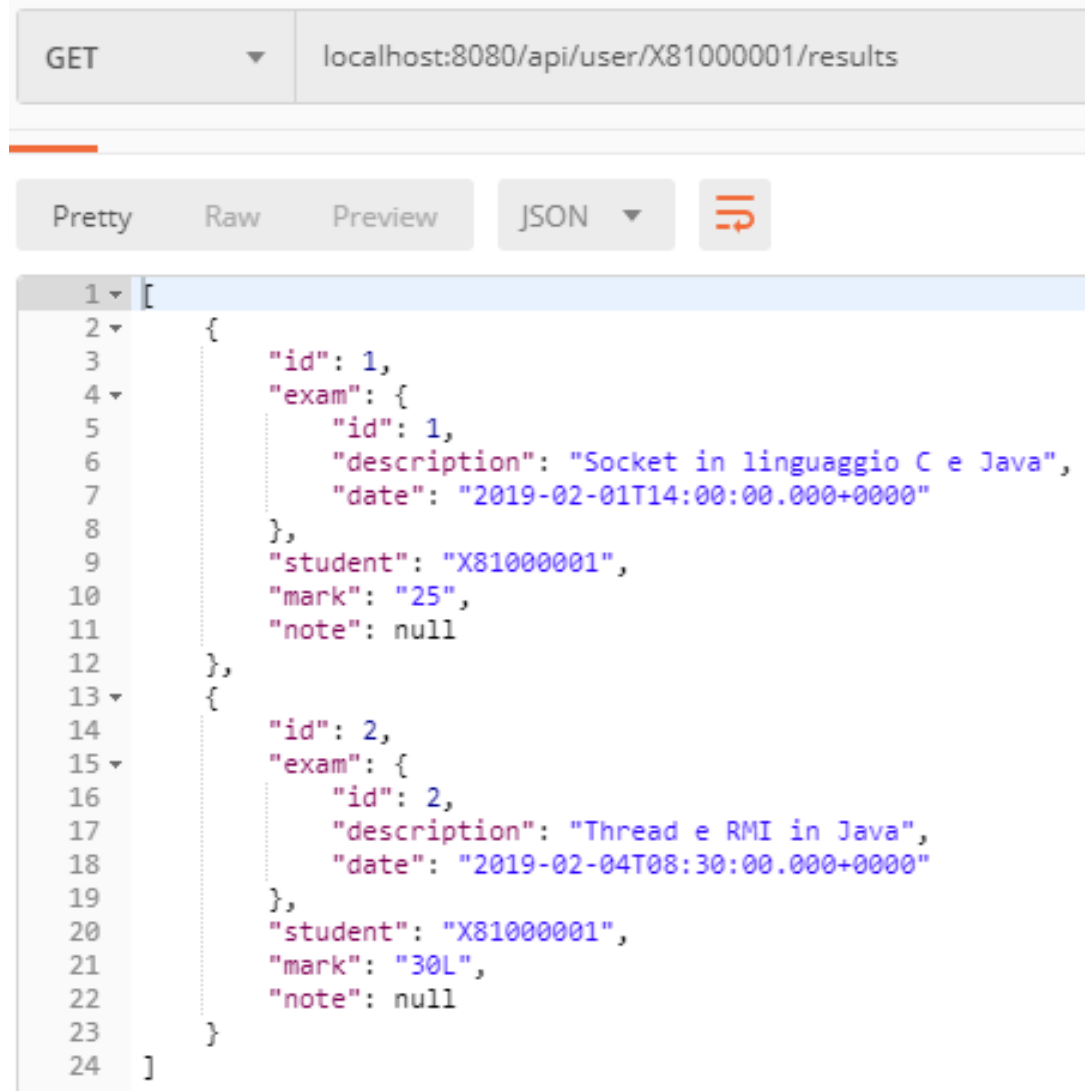


Tutti i risultati di uno studente

Esempio: tutti i risultati dello studente con matricola X81000001



The screenshot shows a REST client interface with a GET request to `localhost:8080/api/user/X81000001/results`. The response is displayed in JSON format, showing two exam results for the student with ID `X81000001`.

```
[{"id": 1, "exam": {"id": 1, "description": "Socket in linguaggio C e Java", "date": "2019-02-01T14:00:00.000+0000"}, "student": "X81000001", "mark": "25", "note": null}, {"id": 2, "exam": {"id": 2, "description": "Thread e RMI in Java", "date": "2019-02-04T08:30:00.000+0000"}, "student": "X81000001", "mark": "30L", "note": null}]
```

Sicurezza

Allo stato attuale tutti possono fare tutto. Non c'è sicurezza.

Implementiamo un sistema di autenticazione e autorizzazione basato sui ruoli (Role-Based Access Control).

Useremo il modulo Spring Security e JWT (<https://jwt.io>).

Tutte le operazioni saranno protette, le richieste non autorizzate saranno negate.

Aggiungere le nuove dipendenze

Apriamo il file pom.xml e aggiungiamo queste due dipendenze dentro il tag dependencies.

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-security</artifactId>  
</dependency>  
  
<dependency>  
  <groupId>io.jsonwebtoken</groupId>  
  <artifactId>jjwt</artifactId>  
  <version>0.9.0</version>  
</dependency>
```

Appena salviamo il file, il nostro IDE scarica e aggiunge le nuove dipendenze al nostro progetto automaticamente.

401 Unauthorized

Solo per aver inserito la dipendenza security, se riavviamo l'applicazione e proviamo a fare una qualsiasi richiesta, otteniamo una risposta con codice di errore 401.

The screenshot shows a web client interface with the following details:

- Method:** GET
- URL:** localhost:8080/api/exams/
- Params:** A table with 4 columns: KEY, VALUE, and DE. The first row contains 'Key' and 'Value'.
- Body:** A tabbed interface with 'Body', 'Cookies (1)', 'Headers (11)', and 'Test Results'. The 'Body' tab is selected, showing a JSON response in 'Pretty' format.
- Status:** 401 Unauthorized (highlighted in yellow)
- JSON Response:**

```
{
  "timestamp": "2018-12-30T17:28:40.718+0000",
  "status": 401,
  "error": "Unauthorized",
  "message": "Unauthorized",
  "path": "/api/exams/"
}
```

RBAC

Creiamo due ruoli per la nostra applicazione: ADMIN e USER.

Un ADMIN ha accesso totale a tutte le operazioni.

Un USER può:

- visualizzare tutti gli esami
- visualizzare solo i propri risultati

Ruoli

Creiamo un enum per descrivere i nostri ruoli e creiamo una nuova Entity

```
// RoleName.java
package portal.domain;

public enum RoleName {
    ROLE_USER, ROLE_ADMIN
}

// Role.java
package portal.domain;

import javax.persistence.Column;

@Entity
@Table(name = "roles")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Enumerated(EnumType.STRING)
    @NaturalId
    @Column(length = 60)
    private RoleName name;

    public Role() {
    }

    public Role(RoleName name) {
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public RoleName getName() {
        return name;
    }

    public void setName(RoleName name) {
        this.name = name;
    }
}
```

Utente

L'entità Utente ha i seguenti campi:

- un id univoco
- una stringa per memorizzare il nome e il cognome
- username
- email
- password
- un insieme di ruoli

L'username e l'email di un utente hanno il vincolo di unicità nella tabella.

Tra Role e User esiste una relazione molti a molti (@ManyToMany) e usiamo un Set<Role> per memorizzare i ruoli di un utente.

Con @JoinTable creiamo una tabella per la relazione molti a molti dove memorizziamo gli id degli utenti e dei ruoli per definire l'appartenenza di un utente ad un ruolo.

L'annotazione @Email impone il formato all'attributo email (deve essere un indirizzo valido).

```
// User.java
package portal.domain;

import java.util.HashSet;

@Entity
@Table(name = "users", uniqueConstraints = { @UniqueConstraint(columnNames = { "username" }),
    @UniqueConstraint(columnNames = { "email" }) })
public class User {

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

    @NotBlank
    @Size(min = 3, max = 50)
    private String name;

    @NotBlank
    @Size(min = 3, max = 50)
    private String username;

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

    @NotBlank
    @Size(min = 6, max = 100)
    private String password;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "user_roles", joinColumns = @JoinColumn(name = "user_id"), inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles = new HashSet<>();

    public User() {
    }

    public User(String name, String username, String email, String password) {
        this.name = name;
        this.username = username;
        this.email = email;
        this.password = password;
    }
}
```

Creiamo i Repository per utenti e ruoli

Aggiungiamo dei metodi che ci serviranno per controllare se durante la registrazione, un username o un indirizzo email è stato già registrato.

```
// RoleRepository.java
package portal.repository;

import java.util.Optional;

@Repository
public interface RoleRepository extends CrudRepository<Role, Long> {
    Optional<Role> findByName(RoleName roleName);
}

// UserRepository.java
package portal.repository;

import java.util.Optional;

@Repository
public interface UserRepository extends CrudRepository<User, Long> {
    Optional<User> findByUsername(String username);

    Boolean existsByUsername(String username);

    Boolean existsByEmail(String email);
}
```


UserPrinciple

È una classe che implementa l'interfaccia UserDetails offerta da Spring. Memorizza le informazioni dell'utente che saranno incapsulate in oggetti Authentication. Usiamo questa classe per memorizzare ulteriori informazioni non inizialmente previste (id, nome ed email).

UserPrinciple (cont)

```
// UserPrinciple.java
package portal.service;

import java.util.Collection;

public class UserPrinciple implements UserDetails {
    private static final long serialVersionUID = 1L;
    private Long id;
    private String name;
    private String username;
    private String email;
    @JsonIgnore
    private String password;
    private Collection<? extends GrantedAuthority> authorities;

    public UserPrinciple(Long id, String name, String username, String email, String password,
        Collection<? extends GrantedAuthority> authorities) {
        this.id = id;
        this.name = name;
        this.username = username;
        this.email = email;
        this.password = password;
        this.authorities = authorities;
    }

    public static UserPrinciple build(User user) {
        List<GrantedAuthority> authorities = user.getRoles().stream()
            .map(role -> new SimpleGrantedAuthority(role.getName().name()))
            .collect(Collectors.toList());

        return new UserPrinciple(user.getId(), user.getName(), user.getUsername(), user.getEmail(), user.getPassword(),
            authorities);
    }

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }

    @Override
    public String getUsername() {
        return username;
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }
}
```

```
@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return true;
}

@Override
public boolean equals(Object o) {
    if (this == o)
        return true;
    if (o == null || getClass() != o.getClass())
        return false;

    UserPrinciple user = (UserPrinciple) o;
    return Objects.equals(id, user.id);
}
```

UserDetailsServiceImpl

Una classe che implementa l'interfaccia UserDetailsService.

Dato un username, restituisce le informazioni dell'utente o genera un'eccezione se l'utente non è stato trovato.

```
// UserDetailsServiceImpl.java
package portal.service;

import org.springframework.beans.factory.annotation.Autowired;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    UserRepository userRepository;

    @Override
    @Transactional
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        User user = userRepository.findByUsername(username).orElseThrow(
            () -> new UsernameNotFoundException("User Not Found with -> username or email : " + username));

        return UserPrincipal.build(user);
    }
}
```

JwtProvider

JwtProvider è una classe di servizio. Può generare un token, validarne uno o restituire l'username associato ad un token.

```
// JwtProvider.java
package portal.security.jwt;

import java.util.Date;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.core.Authentication;
import org.springframework.stereotype.Component;

import io.jsonwebtoken.ExpiredJwtException;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.MalformedJwtException;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.SignatureException;
import io.jsonwebtoken.UnsupportedJwtException;
import portal.service.UserPrinciple;

@Component
public class JwtProvider {
    private static final Logger logger = LoggerFactory.getLogger(JwtProvider.class);
    private final String jwtSecret = "JwtExamPortalAppSecret";
    private final int jwtExpiration = 86400;

    public String generateJwtToken(Authentication authentication) {
        UserPrinciple userPrincipal = (UserPrinciple) authentication.getPrincipal();
        return Jwts.builder().setSubject((userPrincipal.getUsername())).setIssuedAt(new Date())
            .setExpiration(new Date((new Date()).getTime() + jwtExpiration * 1000))
            .signWith(SignatureAlgorithm.HS512, jwtSecret).compact();
    }

    public boolean validateJwtToken(String authToken) {
        try {
            Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(authToken);
            return true;
        } catch (SignatureException e) {
            logger.error("Invalid JWT signature -> Message: {}", e);
        } catch (MalformedJwtException e) {
            logger.error("Invalid JWT token -> Message: {}", e);
        } catch (ExpiredJwtException e) {
            logger.error("Expired JWT token -> Message: {}", e);
        } catch (UnsupportedJwtException e) {
            logger.error("Unsupported JWT token -> Message: {}", e);
        } catch (IllegalArgumentException e) {
            logger.error("JWT claims string is empty -> Message: {}", e);
        }
        return false;
    }

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

JwtAuthEntryPoint

È una classe che implementa l'interfaccia `AuthenticationEntryPoint`. Interviene quando avviene una richiesta che non è autorizzata.

```
// JwtAuthEntryPoint.java
package portal.security.jwt;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

@Component
public class JwtAuthEntryPoint implements AuthenticationEntryPoint {

    private static final Logger logger = LoggerFactory.getLogger(JwtAuthEntryPoint.class);

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException e)
        throws IOException, ServletException {

        logger.error("Unauthorized error. Message - {}", e.getMessage());
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Error -> Unauthorized");
    }
}
```

JwtAuthTokenFilter

È una classe che implementa l'interfaccia `OncePerRequestFilter`. Viene eseguita ad ogni richiesta HTTP. Il metodo `doFilterInternal`:

- Preleva il token dall'intestazione
- Valida il token
- Ottiene l'username dal token validato
- Carica le informazioni dell'utente e costruisce un oggetto `Authentication`
- Assegna l'oggetto appena creato al `Security Context` di Spring

```
// JwtAuthTokenFilter.java
package portal.security.jwt;

import java.io.IOException;

public class JwtAuthTokenFilter extends OncePerRequestFilter {
    private static final Logger logger = LoggerFactory.getLogger(JwtAuthTokenFilter.class);
    @Autowired
    private JwtProvider tokenProvider;
    @Autowired
    private UserDetailsServiceImpl userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        try {
            String jwt = getJwt(request);
            if (jwt != null && tokenProvider.validateJwtToken(jwt)) {
                String username = tokenProvider.getUserNameFromJwtToken(jwt);
                UserDetails 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.error("Can NOT set user authentication -> Message: {}", e);
        }
        filterChain.doFilter(request, response);
    }

    private String getJwt(HttpServletRequest request) {
        String authHeader = request.getHeader("Authorization");
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            return authHeader.replace("Bearer ", "");
        }
        return null;
    }
}
```

WebSecurityConfig

Classe di configurazione.

- Setta l'encoder delle password degli utenti (non vengono salvate password in chiaro)
- Applica i filtri creati precedentemente
- Ad utenti non autenticati permette di utilizzare solo /api/auth per effettuare l'accesso o la registrazione

```
// WebSecurityConfig.java
package portal.security;

import org.springframework.beans.factory.annotation.Autowired;

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    UserDetailsServiceImpl userDetailsService;

    @Autowired
    private JwtAuthEntryPoint unauthorizedHandler;

    @Bean
    public JwtAuthTokenFilter authenticationJwtTokenFilter() {
        return new JwtAuthTokenFilter();
    }

    @Override
    public void configure(AuthenticationManagerBuilder authenticationManagerBuilder) throws Exception {
        authenticationManagerBuilder.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
    }

    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

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

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors().and().csrf().disable().authorizeRequests().antMatchers("/api/auth/**").permitAll().anyRequest()
            .authenticated().and().exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

        http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);
    }
}
```

Controller per l'autenticazione

Prima di costruire il controller per l'autenticazione, creiamo delle classi che definiscono come deve essere una richiesta di login/registrazione e come deve essere la risposta

Struttura richieste di login e registrazione

```
// SignUpForm.java
package portal.message.request;

import java.util.Set;

import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

public class SignUpForm {
    @NotBlank
    @Size(min = 3, max = 50)
    private String name;

    @NotBlank
    @Size(min = 3, max = 50)
    private String username;

    @NotBlank
    @Size(max = 60)
    @Email
    private String email;

    private Set<String> role;

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

    // Getter e Setter
}
```

```
// LoginForm.java
package portal.message.request;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

public class LoginForm {
    @NotBlank
    @Size(min = 3, max = 60)
    private String username;

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

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

Struttura delle risposte

```
// JwtResponse.java
package portal.message.response;

import java.util.Collection;

import org.springframework.security.core.GrantedAuthority;

public class JwtResponse {
    private String token;
    private String type = "Bearer";
    private String username;
    private Collection<? extends GrantedAuthority> authorities;

    public JwtResponse(String accessToken, String username, Collection<? extends GrantedAuthority> authorities) {
        this.token = accessToken;
        this.username = username;
        this.authorities = authorities;
    }

    // Getter e Setter
}
```

```
// ResponseMessage.java
package portal.message.response;

public class ResponseMessage {
    private String message;

    public ResponseMessage(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

AuthController - Login

La parte più importante: il controller dell'autenticazione.

Per il login ci mettiamo in ascolto su `/api/auth/signin` per richieste HTTP di tipo POST. Richiediamo nel body della richiesta un oggetto di `LoginForm` (definito precedentemente) che contiene l'username e la password dell'utente che vuole effettuare il login. Se il login va a buon fine autenticiamo l'utente e generiamo un token JWT che restituiamo nella risposta usando `JwtResponse`. Questo token identifica l'utente ed è necessario in ogni sua successiva richiesta HTTP.

```
// AuthController.java
package portal.controller;

import java.util.HashSet;

@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/auth")
public class AuthController {

    @Autowired
    AuthenticationManager authenticationManager;

    @Autowired
    UserRepository userRepository;

    @Autowired
    RoleRepository roleRepository;

    @Autowired
    PasswordEncoder encoder;

    @Autowired
    JwtProvider jwtProvider;

    @PostMapping("/signin")
    public ResponseEntity<> authenticateUser(@Valid @RequestBody LoginForm loginRequest) {

        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(loginRequest.getUsername(), loginRequest.getPassword()));

        SecurityContextHolder.getContext().setAuthentication(authentication);

        String jwt = jwtProvider.generateJwtToken(authentication);
        UserDetails userDetails = (UserDetails) authentication.getPrincipal();

        return ResponseEntity.ok(new JwtResponse(jwt, userDetails.getUsername(), userDetails.getAuthorities()));
    }
}
```

AuthController - Registrazione

Per la registrazione di un nuovo utente aspettiamo richieste HTTP di tipo POST su `/api/auth/signup`.

Richiediamo nel body un oggetto di tipo `SignUpForm` e controlliamo se esiste già un utente registrato con l'username o l'email passato nel form di registrazione. Se passa questo controllo, registriamo l'utente assegnando il ruolo `USER`. Infine viene restituita una risposta con l'esito della registrazione.

```
@PostMapping("/signup")
public ResponseEntity<> registerUser(@Valid @RequestBody SignUpForm signUpRequest) {
    if (userRepository.existsByUsername(signUpRequest.getUsername())) {
        return new ResponseEntity<>(new ResponseMessage("Fail -> Username is already taken!"),
            HttpStatus.BAD_REQUEST);
    }

    if (userRepository.existsByEmail(signUpRequest.getEmail())) {
        return new ResponseEntity<>(new ResponseMessage("Fail -> Email is already in use!"),
            HttpStatus.BAD_REQUEST);
    }

    User user = new User(signUpRequest.getName(), signUpRequest.getUsername(), signUpRequest.getEmail(),
        encoder.encode(signUpRequest.getPassword()));

    Set<Role> roles = new HashSet<>();

    Role userRole = roleRepository.findByName(RoleName.ROLE_USER)
        .orElseThrow(() -> new RuntimeException("Fail! -> Cause: User Role not find."));
    roles.add(userRole);

    user.setRoles(roles);
    userRepository.save(user);

    return new ResponseEntity<>(new ResponseMessage("User registered successfully!"), HttpStatus.OK);
}
```

Inizializzare il database

Ora che abbiamo tutto pronto, dobbiamo preparare il nostro database. Dobbiamo inserire i ruoli e aggiungere un utente amministratore. Tutti i nuovi utenti avranno il ruolo USER.

Possiamo fare questo creando un nuovo file chiamato «data.sql». Se nel classpath del progetto è presente questo file, Spring esegue le query del file all'avvio dell'applicazione. Aggiungiamo una nuova configurazione nel file application.properties per far importare il file data.sql

```
# application.properties
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://localhost:3306/portal?serverTimezone=Europe/Rome
spring.datasource.username=portaluser
spring.datasource.password=portalpassword
spring.datasource.initialization-mode=always

-- data.sql

-- Inserisci i ruoli se non esistono
INSERT IGNORE INTO roles(name) VALUES ('ROLE_USER'),('ROLE_ADMIN');

-- Se non presente inserisci un utente amministratore e assegna entrambi i ruoli
INSERT IGNORE INTO users(name, username, password, email) VALUES
('Admin', 'admin', '$2a$10$CPC1v9ShoEM3Fc2PJ2NkLuXcau2jUN8k2g5l5hkB0qgMABc4.1hy.', 'admin@email.com');
INSERT IGNORE INTO user_roles(user_id, role_id) VALUES (1,1),(1,2);
```

```
mysql> SELECT U.username, R.name FROM users U, roles R, user_roles UR WHERE UR.user_id=U.id AND UR.role_id=R.id;
+-----+-----+
| username | name |
+-----+-----+
| admin    | ROLE_USER |
| admin    | ROLE_ADMIN |
+-----+-----+
```

Azioni protette

Ora che abbiamo i ruoli, possiamo permettere o vietare l'esecuzione dei metodi in funzione del ruolo dell'utente che richiede l'azione.

Per fare questo usiamo l'annotazione `@PreAuthorize`

Per esempio: vogliamo far aggiungere esami solo agli amministratori, aggiungiamo questa annotazione al metodo del controller per gli esami.

```
@PostMapping("/exams")
@PreAuthorize("hasRole('ADMIN')")
public ResponseEntity<Exam> addExam(@Valid @RequestBody Exam e) throws URISyntaxException {
    if (e.getId() != null) {
        return ResponseEntity.badRequest().build();
    }
    Exam result = examService.addExam(e);
    return ResponseEntity.created(new URI("/api/exams/" + result.getId())).body(result);
}
```

Registrazione di un utente

The screenshot displays a REST client interface for a POST request to `localhost:8080/api/auth/signup`. The 'Body' tab is active, showing a JSON payload with user registration details. Below the request, the 'Test Results' tab shows a successful response with a 200 OK status and a confirmation message.

Request Details:

- Method: POST
- URL: `localhost:8080/api/auth/signup`
- Body Type: JSON (application/json)
- Body Content:

```
1 {  
2   "username": "X81000001",  
3   "password": "ThePassword",  
4   "name": "Mario Rossi",  
5   "email": "mariorossi@email.com"  
6 }
```

Response Details:

- Status: 200 OK
- Body Type: JSON
- Body Content:

```
1 {  
2   "message": "User registered successfully!"  
3 }
```

Login di un utente

POST localhost:8080/api/auth/signin

Params Authorization Headers (1) Body Pre-request Script Tests

none form-data x-www-form-urlencoded raw binary JSON (application/json)

```
1 {
2   "username": "X81000001",
3   "password": "ThePassword"
4 }
```

POST localhost:8080/api/auth/signin Send Save

Pretty Raw Preview JSON

```
1 {
2   "username": "X81000001",
3   "authorities": [
4     {
5       "authority": "ROLE_USER"
6     }
7   ],
8   "accessToken": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJYODUwMDAwMDU0IiwiaXNjaW50dQsImV4cCI6MTU0NjM2MzI4NH0.tIcp-EA-K-sblz4bWpm5r0tzrWcbZMALdokZG0TaY1GxBFSTjbl6C2NAEiN1FvcWYvsJag0x708nkm1Wv7SYfw",
9   "tokenType": "Bearer"
10 }
```


GET su esami con token

Con il login viene restituito il token dell'utente appena connesso. Questo deve far parte dell'intestazione di ogni richiesta per passare il controllo di accesso.

Se il token non è presente viene restituito una risposta con codice 401 Unauthorized.

GET localhost:8080/api/exams

KEY	VALUE	DESCRIPTION
Key	Value	De

Body Cookies (1) Headers (9) Test Results **Status: 401 Unauthorized**

Pretty Raw Preview JSON

```
1 {
2   "timestamp": "2018-12-31T17:26:10.984+0000",
3   "status": 401,
4   "error": "Unauthorized",
5   "message": "Error -> Unauthorized",
6   "path": "/api/exams"
7 }
```

GET localhost:8080/api/exams

KEY	VALUE	DESCRIPTION
Authorization	Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJYOD...	
Key	Value	De

Body Cookies (1) Headers (9) Test Results **Status: 200 OK**

Pretty Raw Preview JSON

```
1 [
2   {
3     "id": 1,
4     "description": "Socket in linguaggio C e Java",
5     "date": "2019-02-01T14:00:00.000+0000"
6   },
7   {
8     "id": 2,
9     "description": "Thread e RMI in Java",
10    "date": "2019-02-04T08:30:00.000+0000"
11  }
12 ]
```

POST su esame

Se lo stesso utente prova a creare un nuovo esame, riceve una risposta con codice 403 perché non è autorizzato a creare nuovi esami.

È importante aggiungere l'annotazione `@PreAuthorize` ai metodi con risorse e operazioni sensibili.

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** localhost:8080/api/exams
- Headers:**
 - Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJYOD...
 - Content-Type: application/json (checked)
- Status:** 403 Forbidden
- Response Body (JSON):**

```
{
  "timestamp": "2018-12-31T17:29:10.195+0000",
  "status": 403,
  "error": "Forbidden",
  "message": "Forbidden",
  "path": "/api/exams"
}
```

POST su esame (cont)

Se il token appartiene ad un utente con il ruolo amministratore, l'operazione viene eseguita normalmente.

POST localhost:8080/api/exams

Params Authorization Headers (2) Body Pre-request Script Tests

	KEY	VALUE	DESC
	Authorization	Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZG...	
<input checked="" type="checkbox"/>	Content-Type	application/json	
	Key	Value	Desc

Body Cookies (1) Headers (10) Test Results Status: 201 Created

Pretty Raw Preview JSON

```
1 {
2   "id": 4,
3   "description": "Spring",
4   "date": "2019-02-01T15:00:00.000+0000"
5 }
```

Risultati dell'utente connesso

Adesso possiamo aggiungere un nuovo metodo al controller dei risultati per far restituire i risultati dell'utente attualmente connesso.

Ricaviamo l'utente attualmente connesso che fa la richiesta utilizzando il SecurityContext. Il SecurityContext possiede l'oggetto Authentication che contiene a sua volta i dati dell'utente. Prendiamo l'username e attraverso il servizio prendiamo i suoi risultati che restituiamo nella risposta HTTP.

```
@GetMapping("/user/results")
public ResponseEntity<List<Result>> getStudentResults() {
    String user = SecurityContextHolder.getContext().getAuthentication().getName();
    List<Result> results = resultService.getStudentResults(user);
    return ResponseEntity.ok().body(results);
}
```

Risultati dell'utente connesso (cont)

The screenshot shows a REST client interface with a GET request to `localhost:8080/api/user/results`. The **Headers** tab is active, showing an `Authorization` header with a Bearer token. The **Body** tab is also active, showing a JSON response with a status of **200 OK**.

KEY	VALUE	DESC
Authorization	Bearer eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJYOD...	
Key	Value	Desc

Body Cookies (1) Headers (9) Test Results Status: 200 OK

Pretty Raw Preview JSON

```
1 [
2   {
3     "id": 1,
4     "exam": {
5       "id": 1,
6       "description": "Socket in linguaggio C e Java",
7       "date": "2019-02-01T14:00:00.000+0000"
8     },
9     "student": "X81000001",
10    "mark": "25",
11    "note": null
12  },
13  {
14    "id": 2,
15    "exam": {
16      "id": 2,
17      "description": "Thread e RMI in Java",
18      "date": "2019-02-04T08:30:00.000+0000"
19    },
20    "student": "X81000001",
21    "mark": "30L",
22    "note": null
23  }
24 ]
```

Build

Per generare il file JAR all'interno della cartella del progetto:

```
$ mvnw clean package
```

Al termine dell'esecuzione dentro la cartella target troveremo il file .jar pronto per essere eseguito

Conclusioni

Adesso ci sono tutti gli strumenti necessari per completare il back-end di questa applicazione. Abbiamo visto: come creare nuove entità; come usare i repository CRUD per le principali query e come aggiungerne altre dichiarando solo un metodo; come rendere disponibili azioni su queste entità usando richieste HTTP e servizi.

Infine abbiamo aggiunto sicurezza all'applicazione implementando un sistema RBAC che fa uso di token JWT.

Adesso non rimane altro da fare che sviluppare il front-end (usando Angular o React per esempio).