DOCUMENTAZIONE TECNICA

Developed by TEAM BRAVO

Davide Cologgi
https://github.com/DavideCologgi
https://it.linkedin.com/in/davide-cologgi-588b62206

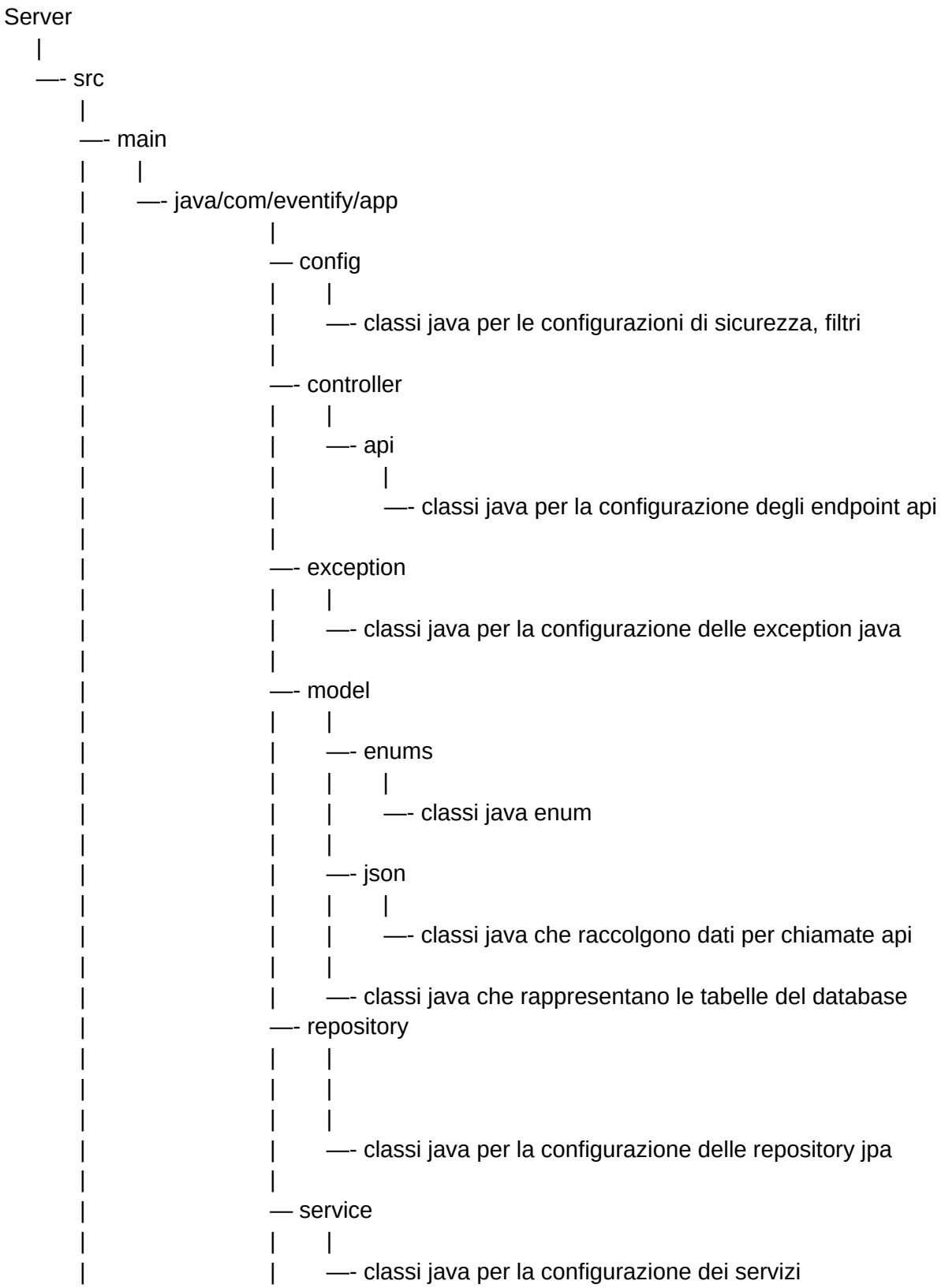
Flaviano Carlucci
https://github.com/fcarlucc
https://www.linkedin.com/in/flaviano-carlucci-1aa816298/

Alessio Buonomo
https://github.com/abuonom
https://www.linkedin.com/in/alessiobuonomo/

Manuele Longo
https://github.com/mlongo03
https://www.linkedin.com/in/manuele-longo-154979243/

Backend

Struttura progetto



```
|                    |
|                    — validator
|                    |       |
|                    |       —- classi java per le configurazioni dei validatori dei dati
|                    |       |     provenienti dalle richieste api
|                    |
|                    — AppApplication.java
|
—- resources
        |
        —- application properties, schema.sql, certificazioni SSL self-signed
```

Configurazioni sicurezza

ApplicationConfig.java

Descrizione Tecnica:

Il file ApplicationConfig rappresenta una configurazione nel contesto di un'applicazione Spring basata su Spring Security. Questa configurazione è responsabile della definizione di vari bean, compresi UserDetailsService, AuthenticationProvider, AuthenticationManager, e PasswordEncoder, che sono essenziali per il sistema di autenticazione e autorizzazione di Spring Security.

Codice Sorgente:

```
package com.eventify.app.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

import com.eventify.app.service.UserService;

import lombok.RequiredArgsConstructor;

@Configuration
@RequiredArgsConstructor
public class ApplicationConfig {

    private final UserService userService;

// Definizione del bean per il servizio di controllo credenziali dell’user
```

```
    @Bean
    public UserDetailsService userDetailsService() {
        return (username) -> userService.findByEmail(username)
                .orElseThrow(() -> new UsernameNotFoundException("User not found"));
    }

// Definizione del bean per l'autenticazione dell'user in particolare si settano le classi dedicate al riconoscimento
//dell'user
    @Bean
    public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
        authProvider.setUserDetailsService(userDetailsService());
        authProvider.setPasswordEncoder(passwordEncoder());
        return authProvider;
    }

// Definizione del bean authenticationManager, bean fondamentale per il riconoscimento dell'user
    @Bean
    public AuthenticationManager  authenticationManager(AuthenticationConfiguration config) throws Exception {
        return config.getAuthenticationManager();
    }

// Definizione del bean per la decodifica della password
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

JwtAuthenticationFilter.java

Descrizione: `JwtAuthenticationFilter` è una classe in un'applicazione Spring che gestisce l'autenticazione basata su JSON Web Token (JWT). Questo filtro è responsabile di verificare i token JWT presenti nelle richieste e, se validi, impostare l'autenticazione per gli utenti.

Utilizzo: Questo filtro viene eseguito per ogni richiesta HTTP e verifica se un token JWT è presente nei cookie. Se la richiesta inizia con `"/api/auth/"`, il filtro la ignora poiché è destinata alle operazioni di autenticazione stesse. In caso contrario, cerca un cookie chiamato `"access_token"`. Se il cookie è presente, il filtro estrae il token JWT e ne estrae l'username. Successivamente, verifica se l'utente associato all'username è autenticato e se il token JWT è valido. Se tutte queste condizioni sono soddisfatte, l'autenticazione dell'utente viene impostata nell'oggetto `SecurityContextHolder`.

Metodo Principale:

- `doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)`: Questo metodo è l'entry point del filtro. Gestisce il flusso principale del filtro.

Metodi di Supporto:

- `loadUserByUsername(String username)`: Carica l'utente associato all'username dall'implementazione di `UserDetailsService`.
- `isTokenValid(String jwt, UserDetails user)`: Verifica la validità del token JWT per l'utente.

Attributi Principali:

- `JwtService jwtService`: Servizio per l'elaborazione dei token JWT.
- `UserDetailsService userDetailsService`: Servizio per il recupero delle informazioni dell'utente.

Diagramma di Flusso:

1. Inizia l'esecuzione del filtro per ogni richiesta.
2. Verifica se la richiesta inizia con `"/api/auth/"`. Se vero, ignora la richiesta e passa al filtro successivo.
3. Cerca il cookie `"access_token"` nella richiesta.
4. Se il cookie non è presente, passa al filtro successivo.
5. Estrae il token JWT dal cookie e ne estrae l'username.
6. Verifica se l'username è valido e se non esiste già un'istanza di autenticazione.
7. Se il token JWT è valido, crea un oggetto di autenticazione e lo imposta nell'oggetto `SecurityContextHolder`.
8. Prosegue con il filtro successivo nella catena di filtri.
9. Fine dell'esecuzione del filtro.

Codice sorgente:

```
package com.eventify.app.config;

import java.io.IOException;

import org.springframework.lang.NonNull;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
```

```
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import org.springframework.web.util.WebUtils;

import com.eventify.app.service.JwtService;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.Cookie;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;

@Component
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {

  private final JwtService jwtService;
  private final UserDetailsService userDetailsService;

  // Sovrascrive il metodo principale per filtrare le richieste
  @Override
  protected void doFilterInternal(@NonNull HttpServletRequest request, @NonNull HttpServletResponse
response, @NonNull FilterChain chain) throws ServletException, IOException {

    // Verifica se la richiesta inizia con "/api/auth/". Se vero, ignora la richiesta.
    if (request.getRequestURI().startsWith("/api/auth/")) {
        chain.doFilter(request, response);
        return;
    }

    // Cerca il cookie chiamato "access_token" nella richiesta.
    Cookie tokenCookie = WebUtils.getCookie(request, "access_token");

    // Se il cookie "access_token" non è presente, prosegui con il filtro successivo.
    if (tokenCookie == null) {
        chain.doFilter(request, response);
        return;
    }

    // Estrae il token JWT dal cookie.
    String jwt = tokenCookie.getValue();
    // Estrae l'username dal token JWT utilizzando il servizio JwtService.
    String username = jwtService.extractUsername(jwt);

    // Verifica se l'username è valido e se non esiste un'istanza di autenticazione.
    if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
        // Carica le informazioni dell'utente tramite il servizio UserDetailsService.
        UserDetails user = this.userDetailsService.loadUserByUsername(username);
        // Verifica se il token JWT è valido utilizzando il servizio JwtService.
        if (jwtService.isTokenValid(jwt, user)) {
            // Crea un oggetto di autenticazione e lo imposta nell'oggetto SecurityContextHolder.
            UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(user,
null, user.getAuthorities());
            authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(authToken);
        }
    }
```

```
    // Prosegue con il filtro successivo nella catena.
    chain.doFilter(request, response);
  }
}
```

RequestLimitFilter.java

Descrizione: `RequestLimitFilter` è una classe di filtro Servlet che limita il numero di richieste da un singolo client in un secondo. Il filtro impedisce il sovraccarico del server bloccando ulteriori richieste da un client che ha superato il limite specificato.

Utilizzo: Questo filtro viene utilizzato per garantire che il server non venga sovraccaricato da un singolo client che effettua troppe richieste in un breve periodo di tempo. Se il cliente supera il limite, riceverà una risposta "429 Too Many Requests" e le ulteriori richieste verranno bloccate fino al successivo intervallo di un secondo.

Metodi Principali:

- `init(FilterConfig filterConfig)`: Inizializza il filtro. In questo caso, il metodo è vuoto poiché il filtro non richiede alcuna inizializzazione specifica.
- `doFilter(ServletRequest request, ServletResponse response, FilterChain chain)`: Questo è il metodo principale del filtro, dove viene gestita la logica del controllo del limite di richieste.
- `destroy()`: Distrugge il filtro. In questo caso, il metodo è vuoto poiché il filtro non richiede alcuna azione di pulizia specifica.

Attributi Principali:

- `MAX_REQUESTS_PER_SECOND`: Costante che specifica il numero massimo di richieste consentite da un singolo cliente in un secondo.
- `requestCountMap`: Una mappa che tiene traccia del numero di richieste inviate da ciascun cliente.
- `requestTimestampMap`: Una mappa che tiene traccia dei timestamp dell'ultima richiesta da parte di ciascun cliente.

Metodo Principale:

- `isRequestLimitExceeded(String clientIP, HttpServletResponse httpResponse)`: Questo metodo verifica se il cliente ha superato il limite di richieste. Se il limite è superato, viene restituita una risposta "429 Too Many Requests". Altrimenti, il metodo aggiorna le mappe dei conteggi e dei timestamp e restituisce `false`.

Diagramma di Flusso:

1. Quando una richiesta arriva al filtro, il clientIP del mittente viene estratto.
2. Il filtro verifica se il limite di richieste è stato superato per il client.
3. Se il limite è superato, viene restituita una risposta "429 Too Many Requests" e il flusso termina.
4. Se il limite non è superato, il filtro permette il passaggio della richiesta alla catena dei filtri successivi e aggiorna i conteggi e i timestamp per il client.
5. Il flusso termina e la risposta viene restituita al client.

Questo filtro è utile per prevenire il sovraccarico del server da parte di client che effettuano troppe richieste in un breve lasso di tempo.

Codice Sorgente:

```
package com.eventify.app.config;

import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.FilterConfig;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import org.springframework.stereotype.Component;

@Component
public class RequestLimitFilter implements Filter {

    private static final int MAX_REQUESTS_PER_SECOND = 30;
    private Map<String, Integer> requestCountMap = new HashMap<>();
    private Map<String, Long> requestTimestampMap = new HashMap<>();

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        // Metodo di inizializzazione del filtro.
        // In questo caso, il metodo è vuoto poiché il filtro non richiede alcuna inizializzazione specifica.
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws
IOException, ServletException {
        // Metodo principale del filtro, dove viene gestita la logica del controllo del limite di richieste.
        String clientIP = request.getRemoteAddr();

        if (isRequestLimitExceeded(clientIP, (HttpServletResponse) response)) {
            // Se il limite di richieste è superato, restituisce una risposta "429 Too Many Requests" e interrompe il
flusso.
```

```
            return;
        } else {
            // Altrimenti, permette il passaggio della richiesta alla catena dei filtri successivi.
            chain.doFilter(request, response);
        }
    }

    @Override
    public void destroy() {
        // Metodo di distruzione del filtro.
        // In questo caso, il metodo è vuoto poiché il filtro non richiede alcuna azione di pulizia specifica.
    }

    private boolean isRequestLimitExceeded(String clientIP, HttpServletResponse httpResponse) {
        try {
            long currentTime = System.currentTimeMillis();

            if (requestCountMap.containsKey(clientIP)) {
                int count = requestCountMap.get(clientIP);
                long lastRequestTime = requestTimestampMap.get(clientIP);
                long timeElapsed = currentTime - lastRequestTime;

                if (count >= MAX_REQUESTS_PER_SECOND && timeElapsed < 1000) {
                    // Se il limite di richieste è superato e il tempo trascorso è inferiore a 1 secondo, restituisce una
risposta "429 Too Many Requests".
                    httpResponse.setStatus(429);
                    httpResponse.getWriter().write("Limite di richieste al secondo superato.");
                    return true;
                } else if (timeElapsed >= 1000) {
                    // Se il tempo trascorso è superiore o uguale a 1 secondo, reimposta il conteggio delle richieste.
                    requestCountMap.put(clientIP, 1);
                    requestTimestampMap.put(clientIP, currentTime);
                } else {
                    // Altrimenti, incrementa il conteggio delle richieste.
                    requestCountMap.put(clientIP, count + 1);
                }
            } else {
                // Se il client non ha ancora effettuato richieste, inizia a tenerne traccia.
                requestCountMap.put(clientIP, 1);
                requestTimestampMap.put(clientIP, currentTime);
            }
            return false;
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
    }
}
```

SqlInjectionFilter.java

Descrizione: `SqlInjectionFilter` è un filtro Spring che ha lo scopo di rilevare e prevenire gli attacchi di SQL injection nelle richieste HTTP in arrivo. Gli attacchi di SQL injection rappresentano un grave rischio per la sicurezza delle applicazioni web,

poiché consentono a un attaccante di eseguire comandi SQL malevoli sul database sfruttando l'input utente.

Utilizzo: Questo filtro viene utilizzato per ispezionare i parametri delle richieste in arrivo e rilevare eventuali tentativi di SQL injection. Se viene rilevato un pattern corrispondente a un attacco di SQL injection, il filtro rifiuta la richiesta e restituisce una risposta "403 Forbidden" con un messaggio di avvertimento.

Metodi Principali:

- `doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)`: Questo è il metodo principale del filtro, dove avviene la rilevazione e la gestione degli attacchi di SQL injection.

Attributi Principali:

- `SQL_INJECTION_PATTERN`: Pattern regex utilizzato per individuare i tentativi di SQL injection nelle richieste. Il pattern corrisponde a varie forme di attacchi SQL injection, inclusi comandi SQL comuni.
- `LOGGER`: Un logger utilizzato per registrare le informazioni sugli attacchi di SQL injection rilevati.

Metodo Principale:

- `doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)`: In questo metodo, il filtro ispeziona i parametri delle richieste in arrivo e cerca corrispondenze con il pattern di SQL injection. Se viene rilevato un attacco, viene registrato un messaggio di log, la richiesta viene rifiutata con uno stato "403 Forbidden", e viene restituito un messaggio di avvertimento.

Diagramma di Flusso:

1. Una richiesta HTTP in arrivo passa attraverso il filtro `SqlInjectionFilter`.
2. Il filtro estrae i parametri dalla richiesta.
3. Per ciascun parametro, il filtro verifica se il suo valore corrisponde al pattern definito in `SQL_INJECTION_PATTERN`.
4. Se viene rilevato un match, il filtro registra un messaggio di avviso tramite il logger e risponde con uno stato "403 Forbidden" e un messaggio di avviso.
5. Se nessun attacco viene rilevato, la richiesta continua a fluire nella catena dei filtri.
6. La risposta finale viene inviata al client.

Questo filtro è un componente critico per la sicurezza delle applicazioni web e aiuta a prevenire potenziali vulnerabilità di SQL injection proteggendo il sistema da attacchi dannosi.

Codice sorgente:

```
package com.eventify.app.config;

import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import org.springframework.http.HttpStatus;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.util.regex.Pattern;
import java.util.Map;
import java.util.regex.Matcher;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Component
public class SqlInjectionFilter extends OncePerRequestFilter {

   // Espressione regolare per rilevare pattern di SQL injection nei parametri delle richieste.
   private static final Pattern SQL_INJECTION_PATTERN = Pattern.compile(
         ".*[;]+.*|.*(--)+.*|.*\\b(ALTER|CREATE|DELETE|DROP|EXEC|INSERT|MERGE|SELECT|UPDATE)\\b.*",
         Pattern.CASE_INSENSITIVE);

   // Logger per registrare eventi di rilevamento di SQL injection.
   private static final Logger LOGGER = LoggerFactory.getLogger(SqlInjectionFilter.class);

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

      // Estrae i parametri dalla richiesta HTTP.
      Map<String, String[]> parameters = request.getParameterMap();

      for (String parameterName : parameters.keySet()) {
         String[] parameterValues = parameters.get(parameterName);
         for (String parameterValue : parameterValues) {

            // Crea un oggetto Matcher per cercare il pattern di SQL injection.
            Matcher matcher = SQL_INJECTION_PATTERN.matcher(parameterValue);
            if (matcher.matches()) {

               // Se viene rilevato un attacco di SQL injection, registra un messaggio di avviso.
               LOGGER.info("SQL injection attack detected! Request: {}", request);

               // Risponde con uno stato "403 Forbidden" e un messaggio di avviso.
               response.setStatus(HttpStatus.FORBIDDEN.value());
               response.getWriter().write("SQL injection attack detected!");

               // Interrompe il flusso della richiesta.
               return;
```

```
            }
        }
    }

    // Se nessun attacco di SQL injection viene rilevato, la richiesta continua a fluire nella catena dei filtri.
    filterChain.doFilter(request, response);
  }
}
```

XssFilter.java

Descrizione: `XssFilter` è una classe di filtro Spring utilizzata per prevenire attacchi di Cross-Site Scripting (XSS) nelle richieste HTTP. Gli attacchi XSS sono un tipo comune di attacco informatico in cui un aggressore cerca di iniettare script dannosi all'interno delle pagine web visualizzate da altri utenti.

Utilizzo: Questo filtro verifica i parametri delle richieste HTTP alla ricerca di potenziali script dannosi o di caratteri speciali che potrebbero essere utilizzati per eseguire script XSS. Se tali elementi sospetti vengono rilevati, il filtro risponde con uno stato "403 Forbidden", impedendo l'elaborazione della richiesta.

Metodi Principali:

- `doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)`: Questo è il metodo principale del filtro, dove avviene la verifica dei parametri delle richieste per rilevare attacchi XSS.
- `containsMaliciousJavaScript(HttpServletRequest request)`: Questo metodo controlla se i parametri delle richieste contengono script JavaScript dannosi o caratteri speciali sospetti.

Attributi Principali:

- Nessun attributo dichiarato in questa classe.

Metodo Principale:

- `doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)`: Questo metodo esegue la verifica dei parametri delle richieste alla ricerca di script XSS o caratteri speciali sospetti. Se vengono rilevati elementi sospetti, il filtro risponde con uno stato "403 Forbidden" e impedisce l'elaborazione della richiesta.

Diagramma di Flusso:

1. Una richiesta HTTP in arrivo passa attraverso il filtro `XssFilter`.
2. Il filtro esegue il metodo `containsMaliciousJavaScript` per verificare se ci sono elementi sospetti nei parametri delle richieste.
3. Se vengono rilevati elementi sospetti, il filtro risponde con uno stato "403 Forbidden" e impedisce l'elaborazione della richiesta.
4. Se nessun elemento sospetto viene rilevato, la richiesta continua a fluire nella catena dei filtri.

Questo filtro è essenziale per proteggere l'applicazione da attacchi di Cross-Site Scripting (XSS), garantendo che i dati in input siano adeguatamente sanificati prima di essere visualizzati o elaborati nella risposta.
Codice sorgente:

```
package com.eventify.app.config;

import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import org.springframework.web.util.HtmlUtils;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Enumeration;

import java.util.Collections;
import java.util.List;

@Component
public class XssFilter extends OncePerRequestFilter {

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

        if (containsMaliciousJavaScript(request)) {
            // Se il filtro rileva script dannosi, risponde con uno stato "403 Forbidden" e termina la catena dei filtri.
            response.setStatus(HttpServletResponse.SC_FORBIDDEN);
            return;
        }
        // Se nessuno script dannoso viene rilevato, la richiesta prosegue nella catena dei filtri.
        filterChain.doFilter(request, response);
    }

    private boolean containsMaliciousJavaScript(HttpServletRequest request) {
        // Questo metodo verifica se i parametri delle richieste contengono script JavaScript dannosi o caratteri
speciali sospetti.

        Enumeration<String> paramNames = request.getParameterNames();
        List<String> paramNameList = Collections.list(paramNames);

        for (String paramName : paramNameList) {
```

```
            String paramValue = request.getParameter(paramName);
            if (paramValue != null) {
                // Controlla se il valore del parametro contiene "<script>" o "alert(".
                if (paramValue.contains("<script>") || paramValue.contains("alert(")) {
                    // Se rileva uno di questi elementi sospetti, restituisce true.
                    return true;
                }
                // Utilizza HtmlUtils.htmlEscape per sanificare il valore del parametro.
                String safeParamValue = HtmlUtils.htmlEscape(paramValue);
                // Se il valore sanificato è diverso dal valore originale, significa che è stato rilevato qualcosa di sospetto.
                if (!safeParamValue.equals(paramValue)) {
                    // Restituisce true in caso di sospetto.
                    return true;
                }
            }
        }
        // Restituisce false se non sono stati rilevati elementi sospetti.
        return false;
    }
}
```

WebConfig.java

Descrizione Tecnica:

Il file `WebConfig` rappresenta una classe di configurazione per la gestione delle impostazioni di Cross-Origin Resource Sharing (CORS) in un'applicazione Spring. Il CORS è una politica di sicurezza che consente o limita le richieste HTTP tra origini diverse, ad esempio tra domini diversi. Questa classe definisce le configurazioni CORS per consentire l'accesso a risorse specifiche da un'origine specifica.

- `@Configuration` e `@EnableWebMvc` sono annotazioni di configurazione Spring che indicano che questa classe è responsabile della configurazione dell'applicazione.
- Il metodo `corsFilter` è annotato con `@Bean` ed è utilizzato per registrare un filtro CORS nella catena dei filtri dell'applicazione. Questo filtro consente di gestire le richieste CORS.
- All'interno del metodo `corsFilter`, viene creato un oggetto `UrlBasedCorsConfigurationSource` denominato `source`, che rappresenta la configurazione CORS basata su URL.
- Viene creato un oggetto `CorsConfiguration` denominato `config` che definisce le regole CORS. Le regole includono:
    - `setAllowCredentials(true)`: Abilita il supporto per le credenziali (cookies) nelle richieste.
    - `addAllowedOrigin("https://localhost:4200")`: Specifica l'origine consentita (in questo caso, "https://localhost:4200").

    - `setAllowedHeaders(Arrays.asList(...))`: Specifica gli header HTTP consentiti nelle richieste.
    - `setAllowedMethods(Arrays.asList(...))`: Specifica i metodi HTTP consentiti nelle richieste (GET, POST, PUT, DELETE).
    - `setMaxAge(3600L)`: Imposta il tempo massimo di memorizzazione nella cache del preflight CORS.
- Infine, l'oggetto `config` viene registrato all'interno dell'oggetto `source` utilizzando il metodo `registerCorsConfiguration("/**", config)` per definire la configurazione CORS per tutte le risorse.
- Viene creato un oggetto `FilterRegistrationBean` denominato `bean` che rappresenta la registrazione del filtro CORS nella catena dei filtri. L'oggetto `CorsFilter` viene inizializzato con l'oggetto `source` creato in precedenza.
- Viene impostato l'ordine di esecuzione del filtro utilizzando il metodo `setOrder(-102)`, che indica l'ordine relativo rispetto ad altri filtri nella catena dei filtri.
- Infine, l'oggetto `bean` viene restituito come risultato del metodo `corsFilter`.

Diagramma di Flusso:

Il diagramma di flusso per questa classe è piuttosto semplice:

1. L'applicazione avvia la configurazione e legge la classe `WebConfig`.
2. Il metodo `corsFilter` viene chiamato e crea un oggetto `UrlBasedCorsConfigurationSource` e un oggetto `CorsConfiguration` con le regole CORS specificate.
3. L'oggetto `config` viene registrato nell'oggetto `source`.
4. Viene creato un oggetto `FilterRegistrationBean` con il filtro CORS.
5. Viene impostato l'ordine di esecuzione del filtro.
6. L'oggetto `bean` viene restituito come risultato del metodo `corsFilter`.
7. Il filtro CORS è ora registrato nell'applicazione e gestirà le richieste CORS secondo le regole specificate.
8. ora Una richiesta HTTP in arrivo passa attraverso il filtro corsFilter.
9. se la richiesta non rispetta i criteri dati dal filtro viene respinta
10. se la richiesta rispetta i criteri prosegue la catena di filtri

In pratica, questa configurazione consente di controllare quali origini possono accedere alle risorse dell'applicazione e quali operazioni HTTP sono consentite da queste origini.

Codice Sorgente:

```
package com.eventify.app.config;

import java.util.Arrays;
```

```
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.web.filter.CorsFilter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@EnableWebMvc
public class WebConfig {

    // Metodo annotato con @Bean per registrare il filtro CORS
    @Bean
    public FilterRegistrationBean<CorsFilter> corsFilter() {
        // Creazione di una configurazione CORS basata su URL
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

        // Creazione di un'istanza di CorsConfiguration per definire le regole CORS
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowCredentials(true);
        config.addAllowedOrigin("https://localhost:4200");
        config.setAllowedHeaders(Arrays.asList(
            HttpHeaders.AUTHORIZATION,
            HttpHeaders.CONTENT_TYPE,
            HttpHeaders.ACCEPT
        ));
        config.setAllowedMethods(Arrays.asList(
            HttpMethod.GET.name(),
            HttpMethod.POST.name(),
            HttpMethod.PUT.name(),
            HttpMethod.DELETE.name()
        ));
        config.setMaxAge(3600L);

        // Registrazione della configurazione CORS per tutte le risorse
        source.registerCorsConfiguration("/**", config);

        // Creazione di un oggetto FilterRegistrationBean per registrare il filtro CORS
        FilterRegistrationBean<CorsFilter> bean = new FilterRegistrationBean<CorsFilter>(new CorsFilter(source));

        // Impostazione dell'ordine di esecuzione del filtro
        bean.setOrder(-102);

        // Restituzione dell'oggetto bean come risultato
        return bean;
    }
}
```

SecurityConfig.java

La classe `SecurityConfig` rappresenta la configurazione di sicurezza per un'applicazione Spring. Questa classe è annotata con `@Configuration`, `@EnableWebSecurity`, e accetta alcune dipendenze necessarie per il suo funzionamento.

Caratteristiche Principali:

1. Filtro di Sicurezza Personalizzati: La configurazione include l'uso di filtri di sicurezza personalizzati come `XssFilter`, `SqlInjectionFilter`, `RequestLimitFilter`, e `JwtAuthenticationFilter`. Questi filtri personalizzati vengono aggiunti alla catena dei filtri e si occupano di attività specifiche, come la protezione da attacchi XSS, iniezione SQL, limitazione delle richieste, e autenticazione JWT.
2. Disabilitazione CSRF: La configurazione disabilita il Cross-Site Request Forgery (CSRF) con il metodo `csrf(AbstractHttpConfigurer::disable)`. Questo è utile quando si sviluppa un'API RESTful in cui le richieste non mantengono uno stato CSRF.
3. Requisiti di Sicurezza: Viene definito un elenco di requisiti di sicurezza per le richieste HTTP tramite il metodo `authorizeHttpRequests((requests) -> ...`. Ad esempio, alcune richieste POST e GET specifiche sono rese pubbliche, mentre le altre richiedono autenticazione.
4. Gestione della Sessione: La sessione dell'utente è gestita con la politica `SessionCreationPolicy.STATELESS`, che indica che non verranno create sessioni utente. Questo è comune nelle applicazioni RESTful stateless.
5. Provider di Autenticazione: È possibile vedere l'utilizzo di un provider di autenticazione (`authenticationProvider`) per l'autenticazione degli utenti. Questo provider personalizzato verrà utilizzato per l'autenticazione JWT.
6. Gestione delle Intestazioni: Le intestazioni delle risposte sono configurate con alcune restrizioni di sicurezza. Ad esempio, viene impostata una `Content-Security-Policy` per limitare l'esecuzione di script alle sole risorse locali. Inoltre, `frameOptions` è configurato con i valori di default per migliorare la sicurezza delle pagine web.
7. Gestione del Logout: La gestione del logout viene configurata con il percorso di logout, gli handler di logout e l'azione da eseguire al successo del logout. Il contesto di sicurezza viene cancellato alla riuscita del logout.

Diagramma di Flusso:

Il diagramma di flusso per questa configurazione è complesso e coinvolge varie fasi:

1. Le richieste HTTP entrano nell'applicazione.
2. Vengono eseguiti i filtri di sicurezza personalizzati (`XssFilter`, `SqlInjectionFilter`, `RequestLimitFilter`) prima della verifica dell'autenticazione.
3. Si verifica che la richiesta abbia un canale sicuro https
4. Le richieste HTTP sono autorizzate in base ai criteri definiti, con alcune richieste che richiedono l'autenticazione.
5. L'autenticazione viene gestita da un provider personalizzato (`authenticationProvider`).
6. Viene aggiunto un filtro di autenticazione JWT (`JwtAuthenticationFilter`) prima della verifica dell'autenticazione.
7. Le risposte vengono elaborate, aggiungendo le intestazioni di sicurezza appropriate e configurando il logout.

Codice sorgente:

```
package com.eventify.app.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import org.springframework.security.web.authentication.logout.LogoutHandler;
import org.springframework.security.config.Customizer;
import org.springframework.security.web.header.writers.StaticHeadersWriter;

import lombok.RequiredArgsConstructor;

@Configuration
@RequiredArgsConstructor
@EnableWebSecurity
public class SecurityConfig {

    // In questa classe di configurazione, vengono definiti vari aspetti della sicurezza dell'applicazione.
    // Questa documentazione fornirà una spiegazione generale dei principali elementi di configurazione.

    // I filtri seguenti vengono iniettati attraverso le dipendenze, inclusi i filtri per la sicurezza:
    // - `XssFilter`
    // - `SqlInjectionFilter`
    // - `RequestLimitFilter`
    // - `JwtAuthenticationFilter`

    // `AuthenticationProvider` viene utilizzato per fornire autenticazione personalizzata.
    // `LogoutHandler` gestisce le azioni di logout.
```

```
    // Definizione del filtro di sicurezza iniziale.
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        // La catena di filtri di sicurezza viene configurata utilizzando l'oggetto `http`.

        return http
                // Aggiunta dei filtri di sicurezza personalizzati prima del filtro di autenticazione di base.
                .addFilterBefore(xssFilter, UsernamePasswordAuthenticationFilter.class)
                .addFilterBefore(sqlInjectionFilter, UsernamePasswordAuthenticationFilter.class)
                .addFilterBefore(requestLimitFilter, UsernamePasswordAuthenticationFilter.class)
                // Disabilita la protezione da attacchi CSRF (Cross-Site Request Forgery).
                .csrf(AbstractHttpConfigurer::disable)
                // Richiede l'uso di connessioni sicure per tutte le richieste.
                .requiresChannel(channel -> channel.anyRequest().requiresSecure())
                // Imposta la politica di creazione della sessione su STATELESS, indicando che non vengono utilizzate
sessioni.
                .sessionManagement(customizer ->
customizer.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                // Definisce le regole di autorizzazione per le richieste HTTP.
                .authorizeHttpRequests((requests) -> requests
                        .requestMatchers(HttpMethod.POST, "/api/auth/**").permitAll()
                        .requestMatchers(HttpMethod.GET, "/teller/subscribe/**").permitAll()
                        .anyRequest().authenticated())
                // Configura il provider di autenticazione personalizzato.
                .authenticationProvider(authenticationProvider)
                // Aggiunge il filtro di autenticazione JWT personalizzato prima del filtro di autenticazione di base.
                .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class)
                // Configura gli header HTTP di sicurezza.
                .headers(headers -> headers
                        // Aggiunge l'header Content-Security-Policy per limitare l'esecuzione di script solo dal dominio
corrente.
                        .addHeaderWriter(new StaticHeadersWriter("Content-Security-Policy", "script-src 'self';"))
                        // Configura le opzioni relative ai frame (nella configurazione predefinita).
                        .frameOptions(Customizer.withDefaults()))
                // Configura le azioni di logout.
                .logout(logout -> logout
                        .logoutUrl("/api/auth/logout")
                        .addLogoutHandler(logoutHandler)
                        .logoutSuccessHandler((request, response, authentication) ->
SecurityContextHolder.clearContext()))
                // Costruisce la catena dei filtri di sicurezza configurata.
                .build();
    }
}
```

Controller

AuthenticationController.java

La classe `AuthenticationController` è una classe controller di un'applicazione Spring che gestisce le richieste relative all'autenticazione degli utenti. Questa classe gestisce vari casi, tra cui la registrazione, il login, l'autenticazione a due fattori (2FA), il ripristino della password e il refresh dei token.

Caratteristiche Principali:

1. Registrazione di un Nuovo Utente: Il metodo `signup` gestisce la registrazione di un nuovo utente. I dettagli dell'utente vengono forniti come parametri nella richiesta HTTP (ad esempio, `firstName`, `email`, ecc.). L'utente deve fornire i dettagli richiesti e le immagini per la registrazione. La richiesta viene elaborata e restituisce una risposta che può contenere un messaggio di errore o una conferma di registrazione avvenuta con successo.
2. Accesso Utente: Il metodo `signIn` gestisce il processo di accesso degli utenti. Prende in input una richiesta di accesso contenente un nome utente (o email) e una password. Se le credenziali sono valide, l'utente riceverà un token di autenticazione per le future richieste.
3. Autenticazione a Due Fattori (2FA): Questo controller gestisce l'autenticazione a due fattori, dove gli utenti devono inserire un codice OTP. Questo è gestito attraverso i metodi `auth2fa` e `refreshauth2fa`. Gli utenti ricevono un codice OTP via email, che devono inserire per completare il processo di autenticazione. Se il codice è corretto, ricevono un token di autenticazione.
4. Gestione del Logout: L'applicazione gestisce anche il logout, ma il metodo `auth2fa` si occupa di eliminare il token di autenticazione al momento del logout.
5. Reset della Password: Gli utenti possono richiedere il reset della password, e questo è gestito attraverso i metodi `resetPasswordRequest` e `resetPasswordRequestValidation`. Gli utenti ricevono un codice OTP via email, che devono utilizzare per reimpostare la password.
6. Refresh dei Token: Il metodo `refreshToken` gestisce il refresh del token di autenticazione per gli utenti che ne sono già in possesso.
7. Validazione dei Dati Utente: Il controller esegue la validazione dei dati dell'utente, inclusa la verifica della forza della password e la corrispondenza tra password e conferma della password.
8. Dipendenze: La classe dipende da molti servizi come `UserService`, `AuthService`, `UserValidator`, `EmailService`, `JwtService`, e altri per eseguire le operazioni di autenticazione.

9. Uso di Model e DTO: Vengono utilizzate classi di modello come `User` e diverse classi di oggetti di trasferimento dati (DTO) come `RegisterRequest`, `LoginRequest`, `AuthenticationResponse`, ecc., per gestire i dati inviati e ricevuti dalle richieste.
10. Validazione: Viene eseguita la validazione dei dati, ad esempio, la conferma della password e la verifica della forza della password. In caso di dati non validi, vengono restituite risposte appropriate.
11. Gestione del Token JWT: Il rinnovo del token di autenticazione (`refreshToken`) e la generazione dei token JWT sono gestiti da `JwtService`.
12. EmailService: Vengono utilizzati servizi email per inviare email all'utente, ad esempio, per la conferma 2FA e il reset della password.

Diagramma di Flusso:

Il flusso delle operazioni all'interno del `AuthenticationController` è complesso e coinvolge diverse azioni, dipendenze e flussi di dati. Ecco un riassunto del flusso principale per alcune delle operazioni chiave:

Registrazione (`signup`):

- I dati vengono inviati come parametri nella richiesta HTTP.
- Viene eseguita la validazione dei dati.
- Se la validazione ha successo, la registrazione dell'utente viene effettuata tramite `AuthService`.
- La risposta viene restituita al client.

Login (`signIn`):

- I dati di accesso vengono inviati come oggetto JSON nella richiesta HTTP.
- Viene eseguita l'autenticazione dell'utente tramite `AuthService`.
- Se l'autenticazione ha successo si da una risposta positiva al client e si reindirizza l’user verso l’autenticazione a due fattori.

Autenticazione a Due Fattori (2FA) (`auth2fa`):

- Viene verificato l'OTP e, se corretto, vengono generati i token JWT e restituiti al client.

Rinnovo del Token di Autenticazione (`refreshToken`):

- Viene inviata una richiesta per il rinnovo del token.
- Il token di aggiornamento viene utilizzato per generare un nuovo token di accesso.
- Il nuovo token di accesso viene restituito al client.

Reset della Password (`resetPasswordRequest`):

- Viene avviato un processo per il reset della password. Viene inviato un OTP all'utente tramite email.
- L'utente invia una richiesta per il reset della password insieme all'OTP.

- L'OTP viene verificato e, se corretto, la password viene reimpostata.

Questo controller gestisce operazioni cruciali per l'autenticazione e la sicurezza degli utenti, e pertanto richiede la massima attenzione nella gestione delle richieste e delle risposte. È inoltre importante notare che il controller utilizza molti servizi di altri componenti (come `authService`, `emailService`, `jwtService`) per l'elaborazione e la validazione dei dati delle richieste.

Codice sorgente:

```
package com.eventify.app.controller.api;

import jakarta.mail.MessagingException;
import jakarta.servlet.http.Cookie;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;

import com.eventify.app.model.User;
import com.eventify.app.model.json.AuthenticationResponse;
import com.eventify.app.model.json.EmailOtp;
import com.eventify.app.model.json.LoginRequest;
import com.eventify.app.model.json.Otp2FA;
import com.eventify.app.model.json.RegisterRequest;
import com.eventify.app.model.json.ResetPasswordRequest;
import com.eventify.app.service.AuthService;
import com.eventify.app.service.EmailService;
import com.eventify.app.service.JwtService;
import com.eventify.app.service.UserService;
import com.eventify.app.validator.UserValidator;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Optional;

import org.springframework.http.ResponseEntity;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@RestController
@RequiredArgsConstructor
public class AuthenticationController {

    private final UserService userService;
    private final AuthService authService;
    private final UserValidator userValidator;
    private final EmailService emailService;
    private final JwtService jwtService;
```

```
// Endpoint per la registrazione di un nuovo utente
  @PostMapping("api/auth/signup")
  public ResponseEntity<String> signup(@RequestParam("firstName") String firstName,
      @RequestParam("lastName") String lastName,
      @RequestParam("dateOfBirth") String dateOfBirth,
      @RequestParam("email") String email,
      @RequestParam("password") String password,
      @RequestParam("confirmPassword") String confirmPassword,
      @RequestParam("photo") MultipartFile photo,
      @RequestParam("checkbox") boolean checkbox) throws Exception {

      if (dateOfBirth.equals(",0000-00-00")) {
         return ResponseEntity.ok("all fields must be filled");
      }
      SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
      Date dateOfBirthh = dateFormat.parse(dateOfBirth);
      RegisterRequest registerRequest = new RegisterRequest();
      registerRequest.setFirstname(firstName);
      registerRequest.setLastname(lastName);
      registerRequest.setDob(dateOfBirthh);
      registerRequest.setEmail(email);
      registerRequest.setPassword(password);
      registerRequest.setConfirmPassword(confirmPassword);
      registerRequest.setProfilePicture(photo);
      registerRequest.setCheckbox(checkbox);
// validazione dei dati da parte del service che nel caso aggiorna il
// database
    return ResponseEntity.ok(authService.signUp(registerRequest));
  }

// Endpoint per l'accesso di un utente
  @PostMapping("api/auth/signin")
  public ResponseEntity<AuthenticationResponse> signIn(@RequestBody LoginRequest loginRequest,
HttpServletRequest request, HttpServletResponse response) throws Exception {
// la validazione delle credenziali e la risposta viene gestita
// dall'auth service
     return ResponseEntity.ok(authService.signIn(loginRequest, request, response));
  }

// Endpoint per verificare se un utente è autenticato
  @PostMapping("/api/authenticate")
  public ResponseEntity<String> authenticate(HttpServletRequest request) {
    return ResponseEntity.ok("You are Authenticate");
  }

// Endpoint per la verifica a due fattori la quale genera i token
// identificativi in caso di successo
  @PostMapping("/api/auth/2FA")
  public ResponseEntity<AuthenticationResponse> auth2fa(@RequestBody Otp2FA otp2FA,
HttpServletResponse response) {
     try {
        Integer otp = Integer.parseInt(otp2FA.otp());
        Optional<User> user = userService.findByOtp(otp);
        if (user.isEmpty()) {
```

```
            return ResponseEntity.ok(AuthenticationResponse.builder().error("Wrong otp
code").accessToken(null).refreshToken(null).expirationDate(null).build());
        }
        user.get().setOtp(null);
        String accessToken = jwtService.generateToken(user.get());
        String refreshToken = jwtService.generateRefreshToken(user.get());
        user.get().setRefreshToken(refreshToken);
        userService.update(user.get().getId(), user.get());
        Cookie accessTokenCookie = new Cookie("access_token", accessToken);
        accessTokenCookie.setHttpOnly(true);
        accessTokenCookie.setPath("/");
        accessTokenCookie.setAttribute("SameSite", "Strict");
        accessTokenCookie.setSecure(true);
        response.addCookie(accessTokenCookie);

        Cookie refreshTokenCookie = new Cookie("refresh_token", refreshToken);
        refreshTokenCookie.setHttpOnly(true);
        refreshTokenCookie.setPath("/");
        refreshTokenCookie.setAttribute("SameSite", "Strict");
        refreshTokenCookie.setSecure(true);
        Date expirationDate = jwtService.extractExpiration(accessToken);
        response.addCookie(refreshTokenCookie);
        return
ResponseEntity.ok(AuthenticationResponse.builder().error(null).userId(user.get().getId()).accessToken(accessTo
ken).refreshToken(refreshToken).expirationDate(expirationDate).build());

    } catch (NumberFormatException e) {
        System.err.println("Failed to convert the string to an integer: " + e.getMessage());
        return ResponseEntity.ok(AuthenticationResponse.builder().error("Wrong otp
code").accessToken(null).refreshToken(null).expirationDate(null).build());
    }
  }

// Endpoint per la rigenererazione del codice OTP
  @PostMapping("/api/auth/refresh-2FA")
  public void refreshauth2fa(@RequestBody EmailOtp emailRefreshOtp) {
    try {
        String secretKey = authService.generateSecretKey();
        int otp = authService.generateOtp(secretKey);

        Optional<User> user = userService.findByEmail(emailRefreshOtp.email());
        user.get().setOtp(otp);
        userService.update(user.get().getId(), user.get());

        emailService.sendRefresh2fa(emailRefreshOtp.email(), otp);
    } catch (MessagingException e) { }
  }

  @PostMapping("/api/auth/signin-failure")
  public void auth2faFailure(@RequestParam("email") String email) {
    try {
        emailService.sendAuthFailure(email);
    } catch (MessagingException e) { }
  }
// Endpoint che genera un codice otp in caso di smarrimento password
  @PostMapping("/api/auth/forgot-password")
```

```
    public ResponseEntity<String> resetPasswordRequest(@RequestBody EmailOtp emailForgotPassword) throws
MessagingException {
        Optional<User> user = userService.findByEmail(emailForgotPassword.email());
        if (user.isEmpty()) {
            return ResponseEntity.ok("Email not found");
        }
        String secretKey = authService.generateSecretKey();
        int otp = authService.generateOtp(secretKey);

        user.get().setOtp(otp);
        userService.update(user.get().getId(), user.get());
        emailService.sendResetPassword(user.get().getEmail(), otp);
        return ResponseEntity.ok(emailForgotPassword.email());
    }

// Endpoint per il reset password se i dati e l'OTP sono validi
    @PostMapping("/api/auth/reset-password")
    public ResponseEntity<String> resetPasswordRequestValidation(@RequestBody ResetPasswordRequest
resetPasswordRequest) throws MessagingException {
        String password = resetPasswordRequest.password();
        String confirmPassword = resetPasswordRequest.confirmPassword();
        try {
            Integer otp = Integer.parseInt(resetPasswordRequest.otp());
            Optional<User> user = userService.findByOtp(otp);
            if (user.isEmpty())
                return ResponseEntity.ok("Wrong otp code");
            if (password.equals(confirmPassword) && password.length() >= 8 &&
userValidator.isStrongPassword(password)) {
                BCryptPasswordEncoder bcrypt = new BCryptPasswordEncoder();
                String EncryptedPwd = bcrypt.encode(password);
                user.get().setPassword(EncryptedPwd);
                user.get().setOtp(null);
                userService.update(user.get().getId(), user.get());
            } else {
                return ResponseEntity.ok("Password must be qual to confirm password and they  must contain at least
8 character, 1 uppercase, 1 special character and 1 digit");
            }
            return ResponseEntity.ok("Reset password done correctly");
        } catch (NumberFormatException e) {
            System.err.println("Failed to convert the string to an integer: " + e.getMessage());
            return ResponseEntity.ok("Wrong otp code");
        }

    }

    @PostMapping("/api/auth/refreshToken")
    public ResponseEntity<AuthenticationResponse> refreshToken(HttpServletRequest request,
HttpServletResponse response) {
        return authService.refreshToken(request, response);
    }
}
```

EventController.java

Il codice fornito è un controller in un'applicazione Spring Boot che gestisce le operazioni relative agli eventi. Questo controller gestisce varie operazioni legate agli eventi, inclusa la creazione, la modifica, la cancellazione e il recupero dei dati degli eventi.

Dipendenze e Inizializzazioni:

> `IEventRepository`, `EmailService`, `EventService`, `UserService`, `EventValidator`, `PhotoService`, `GeocodingService`, `NotificationService`, `EventReminderScheduler`: Questi sono servizi e repository necessari per l'elaborazione delle richieste relative agli eventi. Vengono iniettati tramite l'annotazione `@RequiredArgsConstructor`.

Creazione di un Evento:

- La route `POST /api/create-event/{userId}` consente di creare un nuovo evento. Richiede un'identificazione dell'utente (`userId`) e vari parametri come il titolo, la descrizione, la data, il luogo, la categoria e le foto dell'evento. I dati vengono estratti dai parametri della richiesta e utilizzati per creare un oggetto `EventForm`.
- L'evento viene creato con l'aiuto del servizio `eventService` e i dati vengono inviati all'utente tramite e-mail utilizzando `emailService`.
- La risposta contiene un messaggio di conferma o un messaggio di errore.

Recupero di un Evento per ID:

- La route `GET /api/event/findById/{eventId}` consente di recuperare i dettagli di un evento specifico in base all'ID dell'evento fornito. Se l'evento non viene trovato, restituisce una risposta vuota.
- Se l'evento esiste, vengono recuperati i dati relativi alle immagini, ai partecipanti e ai dettagli dell'evento, e restituiti come oggetto `ResponseGetEvent`.

Recupero di Tutti gli Eventi Validi:

- La route `GET /api/user/all-events` recupera tutti gli eventi validi (non scaduti) e li ordina in base alla data e all'ora. Gli eventi scaduti vengono contrassegnati come scaduti utilizzando il campo `isExpired`.
- Gli eventi validi vengono quindi trasformati in oggetti `ResponseGetEvents` e restituiti come una lista.

Cancellazione di un Evento:

- La route `DELETE /api/event/{eventId}/delete-event/{userId}` consente di cancellare un evento specifico. Richiede l'ID dell'evento e l'ID dell'utente.
- Se l'utente che invia la richiesta è il creatore dell'evento, l'evento viene cancellato. Prima della cancellazione, vengono inviate notifiche via e-mail a tutti i partecipanti dell'evento.
- La risposta contiene un messaggio di conferma o un messaggio di errore.

Mappa degli Eventi:

- La route `GET /api/event/showEventMap` consente di ottenere le coordinate geografiche di un indirizzo fornito. Queste coordinate vengono quindi inviate al frontend per la visualizzazione su una mappa.

Ricerca Eventi per Titolo, Luogo e Categoria:

- Le route `GET /api/event/findBy-title`, `GET /api/event/findBy-place` e `GET /api/event/findBy-category` consentono di cercare eventi in base al titolo, al luogo o alla categoria specificati. Restituiscono gli eventi corrispondenti o una risposta vuota se non ne sono stati trovati.

Registrazione e Annullamento di Partecipazione a un Evento:

- Le route `POST /api/event/{eventId}/register/{userId}` e `DELETE /api/event/{eventId}/unregister/{userId}` consentono agli utenti di registrarsi e annullare la partecipazione a un evento specifico. Vengono gestite le notifiche e le e-mail relative a queste azioni.

Recupero dei Partecipanti all'Evento:

- La route `GET /api/event/{eventId}/participants` consente di ottenere la lista dei partecipanti a un evento specifico. Restituisce una lista di oggetti `ParticipantsResponse` con i nomi e gli ID dei partecipanti.

Recupero degli Eventi Creati da un Utente:

- La route `GET /api/user/{userId}/created-events` consente di ottenere la lista degli eventi creati da un utente specifico.

Recupero degli Eventi a cui un Utente si è Registrato:

- La route `GET /api/user/{userId}/registered-events` consente di ottenere la lista degli eventi a cui un utente specifico si è registrato.

Modifica di un Evento:

- Le route `PUT /api/event/{eventId}/modify-event/{userId}` consentono agli utenti di modificare gli attributi di un evento. Il controllore gestisce la modifica del titolo, della descrizione, della data, del luogo, della categoria e delle immagini dell'evento. Le modifiche vengono applicate solo se l'utente è il creatore dell'evento.
- Le route `PUT /api/event/{eventId}/change-title/{userId}`, `PUT /api/event/{eventId}/change-description/{userId}`, `PUT /api/event/{eventId}/change-date-time/{userId}`, `PUT /api/event/{eventId}/change-place/{userId}`, `PUT /api/event/{eventId}/change-category/{userId}` consentono di modificare attributi specifici dell'evento.
- Vengono gestite notifiche e-mail per informare i partecipanti alle modifiche apportate all'evento.

Codice sorgente:

```
package com.eventify.app.controller.api;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;

import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;

import com.eventify.app.model.*;
import com.eventify.app.model.enums.Categories;
import com.eventify.app.model.json.*;
import com.eventify.app.repository.IEventRepository;
import com.eventify.app.service.*;
import com.fasterxml.jackson.core.JsonProcessingException;

import jakarta.mail.MessagingException;

import lombok.RequiredArgsConstructor;

@RestController
@RequiredArgsConstructor
```

```
public class EventController {

    // Iniezione delle dipendenze attraverso il costruttore
    private final IEventRepository eventRepository;
    private final EmailService emailService;
    private final EventService eventService;
    private final UserService userService;
    private final EventValidator eventValidator;
    private final PhotoService photoService;
    private final GeocodingService geocodingService;
    private final NotificationService notificationService;
    private final EventReminderScheduler eventReminderScheduler;

    // Gestisce la creazione di un evento
    @PostMapping("/api/create-event/{userId}")
    public ResponseEntity<String> createEvent(@PathVariable Long userId,
        @RequestParam("title") String title,
        @RequestParam("description") String description,
        @RequestParam("dateTime") String dateTime,
        @RequestParam("place") String place,
        @RequestParam("category") Categories category,
        @RequestParam("photos") List<MultipartFile> photos) {

        // Creazione di un oggetto EventForm per contenere i dati dell'evento
        EventForm eventRequest = EventForm.builder()
            .category(category)
            .dateTime(dateTime)
            .description(description)
            .place(place)
            .title(title)
            .photos(photos)
            .build();

        try {
            // Recupera l'utente e crea un evento
            Optional<User> user = userService.getById(userId);
            List<Object> response = eventService.createEvent(userId, eventRequest);

            // Invia una conferma via email all'utente
            emailService.sendCreationEventConfirm(user.get().getEmail(), eventRequest.getTitle());

            return ResponseEntity.status(HttpStatus.OK).body((String)response.get(0));
        } catch (Exception e) {
            System.out.println(e.getMessage());
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Error creating the event: " +
e.getMessage());
        }
    }

    // Gestisce la ricerca di un evento per ID
    @GetMapping("/api/event/findById/{eventId}")
    public ResponseEntity<ResponseGetEvent> findEventbyId(@PathVariable Long eventId) {

        Optional<Event> event = eventRepository.findById(eventId);

        if (event.isEmpty()) {
```

```
            return ResponseEntity.ok(null);
        }

        List<String> images = new ArrayList<>();

        for (int i = 0; i < event.get().getPhotos().size(); i++) {
            images.add("/api/download/" + event.get().getPhotos().get(i).getId());
        }

        List<ParticipantsResponse> participants = new ArrayList<>();
        List<User> participant = event.get().getParticipants();

        for (int i = 0; i < participant.size(); i++) {
            participants.add(ParticipantsResponse.builder()
                .name(participant.get(i).getFirstname() + " " + participant.get(i).getLastname())
                .id(participant.get(i).getId())
                .build());
        }

        ResponseGetEvent responseGetEvent = ResponseGetEvent.builder()
            .id(event.get().getId())
            .title(event.get().getTitle())
            .address(event.get().getPlace())
            .category(event.get().getCategory().name())
            .date(event.get().getDateTime().toString())
            .description(event.get().getDescription())
            .imageURL(images)
            .participants(participants)
            .build();

        return ResponseEntity.ok(responseGetEvent);
    }

    // Gestisce la ricerca di tutti gli eventi
    @GetMapping("/api/user/all-events")
    public ResponseEntity<List<ResponseGetEvents>> getAllEvents() {

        List<ResponseGetEvents> responseGetEvents = new ArrayList<>();
        List<Event> events = eventService.getAllEvents();

        events.sort(Comparator.comparing(Event::getDateTime));

        List<Event> validEvents = new ArrayList<>();
        LocalDateTime currentDateTime = LocalDateTime.now();

        for (Event event : events) {
            if (!event.getIsExpired() && !event.getDateTime().isBefore(currentDateTime)) {
                validEvents.add(event);
            } else {
                event.setExpired(true);
                eventService.updateEvent(event);
            }
        }
        for (Event event : validEvents) {
            responseGetEvents.add(new ResponseGetEvents(event.getId(), event.getTitle(),
                event.getCategory().name(), event.getDescription(),
```

```
                event.getPlace(), event.getDateTime().toString()
                , "/api/download/" + event.getPhotos().get(0).getId()));
        }
        return ResponseEntity.ok(responseGetEvents);
    }

    // Gestisce la cancellazione di un evento
    @DeleteMapping("/api/event/{eventId}/delete-event/{userId}")
    public ResponseEntity<String> deleteEvent(@PathVariable Long eventId, @PathVariable Long userId) throws
JsonProcessingException {
        Optional<Event> eventOptional = eventService.getEventById(eventId);
        Optional<User> userOptional = userService.getById(userId);

        if (eventOptional.isPresent() && userOptional.isPresent()) {
            Event event = eventOptional.get();
            List<User> participants = event.getParticipants();
            String title = event.getTitle();
            User user = userOptional.get();
            if (event.getCreator().getId().equals(userId)) {
                eventReminderScheduler.unscheduleEventReminders(event);
                eventService.deletePhotosByEvent(eventService.getEventById(eventId).get());
                eventService.deleteEventById(eventId);
                try {
                    // qui si invia la notifica al creatore evento e ai partecipanti
                    notificationService.createNotification(userId, null, "Event correctly deleted, named : ", title);
                    notificationService.sendNotificationToUserId(userId);
                    for (User participant: participants) {
                        notificationService.createNotification(participant.getId(), null, "The event you were subscribed has
been deleted, named : ", title);
                        notificationService.sendNotificationToUserId(participant.getId());
                    }
                    emailService.sendEventAbortedAdvice(user.getEmail(), event.getTitle());
                    return ResponseEntity.ok("All users and the event deleted.");
                } catch (MessagingException e) {
                    return ResponseEntity.badRequest().body("Mail not sent.");
                }
            } else {
                return ResponseEntity.badRequest().body("User isn't the creator.");
            }
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // Gestisce la visualizzazione di una mappa per un evento
    @GetMapping("/api/event/showEventMap")
    public ResponseEntity<String> showEventMap(@RequestParam("address") String eventAddress, Model model)
{
        double[] coordinates = geocodingService.getCoordinatesFromAddress(eventAddress);
        if (coordinates != null) {
            double latitude = coordinates[0];
            double longitude = coordinates[1];

            model.addAttribute("EVENT_LATITUDE", latitude);
            model.addAttribute("EVENT_LONGITUDE", longitude);
```

```
            return ResponseEntity.ok("Localization of the event available.");
        } else {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Coordinates not found.");
    }

    // Gestisce la ricerca di un evento per titolo
    @GetMapping("/api/event/findBy-title")
    public ResponseEntity<Optional<Event>> findEventsByTitle(@RequestParam String title) {
        Optional<Event> event = eventService.findByTitle(title);
        if (!event.isEmpty()) {
            return ResponseEntity.ok(event);
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // Gestisce la ricerca di eventi per luogo
    @GetMapping("/api/event/findBy-place")
    public ResponseEntity<List<Event>> findEventsByPlace(@RequestParam String place) {
        List<Event> events = eventService.findEventsByPlace(place);
        if (!events.isEmpty()) {
            return ResponseEntity.ok(events);
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // Gestisce la ricerca di eventi per categoria
    @GetMapping("/api/event/findBy-category")
    public ResponseEntity<List<Event>> findEventsByCategory(@RequestParam Categories category) {
        List<Event> events = eventService.findEventsByCategory(category);
        if (!events.isEmpty()) {
            return ResponseEntity.ok(events);
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // Gestisce la registrazione a un evento
    @PostMapping("/api/event/{eventId}/register/{userId}")
    public ResponseEntity<String> registerForEvent(@PathVariable Long eventId, @PathVariable Long userId)
throws JsonProcessingException {
        Optional<Event> eventOptional = eventService.getEventById(eventId);
        Optional<User> userOptional = userService.getById(userId);

        if (eventOptional.isPresent() && userOptional.isPresent()) {
            Event event = eventOptional.get();
            User user = userOptional.get();
            List<User> participants = event.getParticipants();

            participants.add(user);
            event.setParticipants(participants);
            eventService.updateEvent(event);
            eventReminderScheduler.scheduleEventReminders(eventService.getEventById(eventId).get());

            try {
```

```
            // qui si invia la notifica al creatore
            notificationService.createNotification(userId, eventId, "Correctly subscribed for the event! ", null);
            notificationService.sendNotificationToUserId(userId);
            notificationService.createNotification(userId, eventId, "Subscribed for creator", null);

notificationService.sendNotificationToUserId(eventService.getEventById(eventId).get().getCreator().getId());
            emailService.sendRegisterEventConfirm(user.getEmail(), event.getTitle());
        } catch (MessagingException e) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Mail not sent.");
        }

        return ResponseEntity.status(HttpStatus.CREATED).body("User is registered for the event.");
    }
    return ResponseEntity.notFound().build();
  }

  // Gestisce la disiscrizione da un evento
  @DeleteMapping("/api/event/{eventId}/unregister/{userId}")
  public ResponseEntity<String> unregisterFromEvent(@PathVariable Long eventId, @PathVariable Long userId)
throws JsonProcessingException {
    Optional<Event> eventOptional = eventService.getEventById(eventId);
    Optional<User> userOptional = userService.getById(userId);

    if (eventOptional.isPresent()) {
      Event event = eventOptional.get();
      User user = userOptional.get();
      List<User> participants = event.getParticipants();

      if (participants.size() != 0) {
        boolean removed = participants.removeIf(participant -> participant.getId().equals(userId));

        if (removed) {
          event.setParticipants(participants);
          eventService.updateEvent(event);
          eventReminderScheduler.unscheduleEventReminders(event);

          try {
            // qui si invia la notifica al creatore
            notificationService.createNotification(userId, eventId, "Correctly unsubscribed for the event! ", null);
            notificationService.sendNotificationToUserId(userId);
            notificationService.createNotification(userId, eventId, "Unsubscribed for creator", null);

notificationService.sendNotificationToUserId(eventService.getEventById(eventId).get().getCreator().getId());
            emailService.sendUnregisterEventConfirm(user.getEmail(), event.getTitle());
          } catch (MessagingException e) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Mail not sent.");
          }
          return ResponseEntity.ok("User has been unregistered for the event");
        }
      }
      return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("User isn't registered for the event.");
    }
    return ResponseEntity.notFound().build();
  }

  // Gestisce il recupero dei partecipanti a un evento
```

```
@GetMapping("/api/event/{eventId}/participants")
public ResponseEntity<List<ParticipantsResponse>> getEventParticipants(@PathVariable Long eventId) {
    Optional<Event> eventOptional = eventService.getEventById(eventId);

    if (eventOptional.isPresent()) {
        Event event = eventOptional.get();
        List<User> participants = event.getParticipants();
        if (participants.size() == 0) {
            return ResponseEntity.ok(null);
        }
        List<ParticipantsResponse> names = new ArrayList<>();
        for (User user: participants) {
            names.add(ParticipantsResponse.builder()
            .name(user.getFirstname() + " " + user.getLastname())
            .id(user.getId())
            .build());
        }
        return ResponseEntity.ok(names);
    }
    return ResponseEntity.ok(null);
}

// Gestisce il recupero degli eventi creati da un utente
@GetMapping("/api/user/{userId}/created-events")
public ResponseEntity<List<Event>> getCreatedEvents(@PathVariable Long userId) {
    Optional<User> userOptional = userService.getById(userId);

    if (userOptional.isPresent()) {
        User user = userOptional.get();
        List<Event> createdEvents = eventService.getEventsCreatedByUser(user);
        return ResponseEntity.ok(createdEvents);
    }
    return ResponseEntity.notFound().build();
}

// Gestisce il recupero degli eventi a cui un utente si è registrato
@GetMapping("/api/user/{userId}/registered-events")
public ResponseEntity<List<Event>> getRegisteredEvents(@PathVariable Long userId) {
    Optional<User> userOptional = userService.getById(userId);

    if (userOptional.isPresent()) {
        User user = userOptional.get();
        List<Event> registeredEvents = eventService.getEventsRegisteredByUser(user);
        return ResponseEntity.ok(registeredEvents);
    }

    return ResponseEntity.notFound().build();
}

// Gestisce la modifica di un evento
@PutMapping("/api/event/{eventId}/modify-event/{userId}")
public ResponseEntity<String> modifyEvent(@PathVariable Long eventId, @PathVariable Long userId,
    @RequestParam("title") String title,
    @RequestParam("description") String description,
    @RequestParam("dateTime") String dateTime,
    @RequestParam("place") String place,
```

```
@RequestParam("category") Categories category,
@RequestParam("photos") List<MultipartFile> photos) {
Optional<Event> eventOptional = eventService.getEventById(eventId);
Optional<User> userOptional = userService.getById(userId);

EventForm eventRequest = EventForm.builder()
.category(category)
.dateTime(dateTime)
.description(description)
.place(place)
.title(title)
.photos(photos)
.build();

if (eventOptional.isPresent()) {
   Event event = eventOptional.get();
   User user = userOptional.get();
   if (event.getCreator().equals(user)) {
      String check = eventValidator.isFormValid(eventRequest);
      eventReminderScheduler.unscheduleEventReminders(event);
      if (check == null) {
         DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss.SSS");
         try {
            LocalDateTime dateTimeUpdate = LocalDateTime.parse(dateTime, formatter);
            Event eventToUpdate = eventService.getEventById(eventId).get();
            eventToUpdate.setCategory(category);
            eventToUpdate.setDateTime(dateTimeUpdate);
            eventToUpdate.setDescription(description);
            eventToUpdate.setTitle(title);
            eventService.deletePhotosByEvent(eventToUpdate);
            for (MultipartFile photo : photos) {
               Photo pic = photoService.uploadPhoto(photo);
               pic.setEvent(eventToUpdate);
               photoService.create(pic);

               if (eventToUpdate.getPhotos() == null) {
                  eventToUpdate.setPhotos(new ArrayList<>());
               }
               eventToUpdate.getPhotos().add(pic);
            }
            eventService.updateEvent(eventToUpdate);
            eventReminderScheduler.scheduleEventReminders(event);
         } catch (Exception e) {
            System.out.println(e.getMessage());
         }
      } else {
         return ResponseEntity.ok("Not valid updates: " + check);
      }
      try {
         for (User participant : event.getParticipants()) {
            emailService.sendChangesAdviseAboutEvent(participant.getEmail(), event.getTitle());
            notificationService.createNotification(userId, eventId, "Updated event named: ", null);
         }
      } catch (MessagingException e) {
         e.printStackTrace();
         return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Sending email error.");
```

```
            }
            return ResponseEntity.ok("Event successfully modified.");
        } else {
            return ResponseEntity.ok("User not recognized as the creator.");
        }
    }
    return ResponseEntity.notFound().build();
}

// Gestisce la modifica del titolo di un evento
@PutMapping("/api/event/{eventId}/change-title/{userId}")
public ResponseEntity<String> changeEventTitle(@PathVariable Long eventId, @PathVariable Long userId,
@RequestParam String newTitle) {
    Optional<Event> eventOptional = eventService.getEventById(eventId);
    Optional(User> userOptional = userService.getById(userId);

    if (eventOptional.isPresent()) {
        Event event = eventOptional.get();
        User user = userOptional.get();
        if (event.getCreator().equals(user)) {
            event.setTitle(newTitle);
            eventService.updateEvent(event);

            try {
                for (User participant : event.getParticipants()) {
                    emailService.sendChangesAdviseAboutEvent(participant.getEmail(), event.getTitle());
                }
            } catch (MessagingException e) {
                e.printStackTrace();
                return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Sending email error.");
            }
            return ResponseEntity.ok("Title of the event successfully modified.");
        } else {
            return ResponseEntity.ok("User not recognized as the creator.");
        }
    }
    return ResponseEntity.notFound().build();
}

// Gestisce la modifica della descrizione di un evento
@PutMapping("/api/event/{eventId}/change-description/{userId}")
public ResponseEntity<String> changeEventDescription(@PathVariable Long eventId, @PathVariable Long
userId, @RequestParam String newDescription) {
    Optional<Event> eventOptional = eventService.getEventById(eventId);
    Optional<User> userOptional = userService.getById(userId);

    if (eventOptional.isPresent()) {
        Event event = eventOptional.get();
        User user = userOptional.get();
        if (event.getCreator().equals(user)) {
            event.setDescription(newDescription);
            eventService.updateEvent(event);

            try {
                for (User participant : event.getParticipants()) {
                    emailService.sendChangesAdviseAboutEvent(participant.getEmail(), event.getTitle());
```

```
                }
            } catch (MessagingException e) {
                e.printStackTrace();
                return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Sending email error.");
            }
            return ResponseEntity.ok("Event description successfully modified.");
        } else {
            return ResponseEntity.ok("User not recognized as the creator.");
        }
    }
    return ResponseEntity.notFound().build();
}

// Gestisce la modifica della data e dell'ora di un evento
@PutMapping("/api/event/{eventId}/change-date-time/{userId}")
public ResponseEntity<String> changeEventDateTime(@PathVariable Long eventId, @PathVariable Long
userId, @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.DATE_TIME) LocalDateTime
newDateTime) {
    Optional<Event> eventOptional = eventService.getEventById(eventId);
    Optional<User> userOptional = userService.getById(userId);

    if (eventOptional.isPresent()) {
        Event event = eventOptional.get();
        User user = userOptional.get();
        if (event.getCreator().equals(user)) {
            event.setDateTime(newDateTime);
            eventService.updateEvent(event);

            try {
                for (User participant : event.getparticipants()) {
                    emailService.sendChangesAdviseAboutEvent(participant.getEmail(), event.getTitle());
                }
            } catch (MessagingException e) {
                e.printStackTrace();
                return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Sending email error.");
            }
            return ResponseEntity.ok("Date and time successfully modified.");
        } else {
            return ResponseEntity.ok("User not recognized as the creator.");
        }
    }
    return ResponseEntity.notFound().build();
}

// Gestisce la modifica del luogo di un evento
@PutMapping("/api/event/{eventId}/change-place/{userId}")
public ResponseEntity<String> changeEventPlace(@PathVariable Long eventId, @PathVariable Long userId,
@RequestParam String place) {
    Optional<Event> eventOptional = eventService.getEventById(eventId);
    Optional<User> userOptional = userService.getById(userId);

    if (eventOptional.isPresent()) {
        Event event = eventOptional.get();
        User user = userOptional.get();
        if (event.getCreator().equals(user)) {
            event.setPlace(place);
```

```
            eventService.updateEvent(event);

            try {
                for (User participant : event.getParticipants()) {
                    emailService.sendChangesAdviseAboutEvent(participant.getEmail(), event.getTitle());
                }
            } catch (MessagingException e) {
                e.printStackTrace();
                return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Sending email error.");
            }
            return ResponseEntity.ok("New place for the event successfully updated.");
        } else {
            return ResponseEntity.ok("User not recognized as the creator.");
        }
    }
    return ResponseEntity.notFound().build();
  }

  // Gestisce la modifica della categoria di un evento
  @PutMapping("/api/event/{eventId}/change-category/{userId}")
  public ResponseEntity<String> changeEventCategory(@PathVariable Long eventId, @PathVariable Long userId,
@RequestParam Categories newCategory) {
    Optional<Event> eventOptional = eventService.getEventById(eventId);
    Optional<User> userOptional = userService.getById(userId);

    if (eventOptional.isPresent()) {
        Event event = eventOptional.get();
        User user = userOptional.get();
        if (event.getCreator().equals(user)) {
            event.setCategory(newCategory);
            eventService.updateEvent(event);

            try {
                for (User participant : event.getParticipants()) {
                    emailService.sendChangesAdviseAboutEvent(participant.getEmail(), event.getTitle());
                }
            } catch (MessagingException e) {
                e.printStackTrace();
                return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Sending email error.");
            }
            return ResponseEntity.ok("Event category successfully modified.");
        } else {
            return ResponseEntity.ok("User not recognized as the creator.");
        }
    }
    return ResponseEntity.notFound().build();
  }
}
```

FilterController.java

Questo file rappresenta un controller in un'applicazione Java basata su Spring che gestisce le richieste relative ai filtri degli eventi.

- `@RestController` e `@RequiredArgsConstructor`: Queste annotazioni sono fornite da Project Lombok. `@RestController` indica che questa classe è un controller Spring, che risponde alle richieste HTTP. `@RequiredArgsConstructor` genera un costruttore con tutti i campi richiesti per l'iniezione delle dipendenze. I campi `filterService` e `eventService` verranno iniettati automaticamente attraverso il costruttore.
- `FilterService` e `EventService`: Queste dipendenze vengono iniettate nel costruttore attraverso `@RequiredArgsConstructor`. `FilterService` e `EventService` sembrano essere servizi utilizzati per applicare i filtri agli eventi.
- `@PostMapping("/filter/{userId}")`: Questa annotazione definisce la mappatura dell'endpoint REST. L'endpoint sarà accessibile tramite richieste HTTP POST all'indirizzo `/filter/{userId}`, dove `{userId}` è una variabile dinamica.
- `public ResponseEntity<List<ResponseGetEvents>> findEventsByFilter(@PathVariable Long userId, @RequestBody FilterForm filterForm)`: Questo è il metodo principale del controller. Prende un parametro `userId` dall'URL tramite `@PathVariable` e un oggetto `filterForm` dalla richiesta HTTP POST tramite `@RequestBody`. Restituirà un oggetto `ResponseEntity` che contiene una lista di eventi filtrati.
- All'interno del metodo, vengono eseguite una serie di operazioni di filtro sugli eventi:
    - Vengono ottenuti tutti gli eventi chiamando `eventService.getAllEvents()`.
    - Viene creata una lista `filteredEvents` inizializzata con tutti gli eventi.
    - Viene controllato il valore di `filterForm.typeEventPage()` per vedere se gli eventi dovrebbero essere filtrati in base alla pagina dell'utente (ad esempio, "my-events" o "registered-events") e vengono applicati i filtri appropriati.
    - Vengono applicati filtri basati su titolo, luogo, data di inizio, data di fine e categoria se i rispettivi campi in `filterForm` non sono vuoti o nulli.
    - Viene eseguito un ulteriore filtro per gli eventi con una data compresa tra `dateStart` e `dateEnd`.
    - Gli eventi filtrati vengono ordinati in base alla data e all'ora tramite `filteredEvents.sort(Comparator.comparing(Event::getDateTime))`.
    - Viene creata una lista `validEvents` contenente eventi non scaduti.
    - Gli eventi scaduti vengono contrassegnati come "scaduti" e vengono aggiornati utilizzando `event.setExpired(true)` e `eventService.updateEvent(event)`.
    - Viene popolata la lista `responseGetEvents` con i dettagli degli eventi filtrati e non scaduti.

- Infine, viene restituita una risposta HTTP con la lista di eventi filtrati in un oggetto `ResponseEntity`.

Codice sorgente:

```
package com.eventify.app.controller.api;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.eventify.app.model.Event;
import com.eventify.app.model.enums.Categories;
import com.eventify.app.model.json.FilterForm;
import com.eventify.app.model.json.ResponseGetEvents;
import com.eventify.app.service.EventService;
import com.eventify.app.service.FilterService;

import lombok.RequiredArgsConstructor;

@RestController
@RequiredArgsConstructor
public class FilterController {

    private final FilterService filterService;
    private final EventService eventService;

//endpoint per il filtraggio degli eventi
    @PostMapping("/filter/{userId}")
    public ResponseEntity<List<ResponseGetEvents>> findEventsByFilter(@PathVariable Long userId,
@RequestBody FilterForm filterForm) {
        // Otteniamo tutti gli eventi
        List<Event> events = eventService.getAllEvents();
        // Creiamo una copia della lista di eventi per l'applicazione dei filtri
        List<Event> filteredEvents = new ArrayList<>(events);

        // Applichiamo i filtri in base al tipo di pagina degli eventi (ad es. "my-events" o "registered-events")
        if (filterForm.typeEventPage().equals("/my-events")) {
            filteredEvents.retainAll(filterService.findEventsByMyEvents(userId, events));
        }

        if (filterForm.typeEventPage().equals("/registered-events")) {
            filteredEvents.retainAll(filterService.findEventsByRegisteredEvents(userId, events));
        }

        // Filtriamo in base al titolo dell'evento
        if (filterForm.title() != null && !filterForm.title().isEmpty()) {
            filteredEvents.retainAll(filterService.findEventsByTitle(filterForm.title(), events));
        }

        // Filtriamo in base al luogo dell'evento
```

```
        if (filterForm.place() != null && !filterForm.place().isEmpty()) {
            filteredEvents.retainAll(filterService.findEventsByPlace(filterForm.place(), events));
        }

        // Filtriamo in base alla data di inizio dell'evento
        if (filterForm.dateStart() != null && !filterForm.dateStart().isEmpty()) {
            filteredEvents.retainAll(filterService.findEventsByDateStart(filterForm.dateStart(), events));
        }

        // Filtriamo in base alla data di fine dell'evento
        if (filterForm.dateEnd() != null && !filterForm.dateEnd().isEmpty()) {
            filteredEvents.retainAll(filterService.findEventsByDateEnd(filterForm.dateEnd(), events));
        }

        // Filtriamo in base a un intervallo di date
        if (filterForm.dateStart() != null && !filterForm.dateStart().isEmpty() &&
            filterForm.dateEnd() != null && !filterForm.dateEnd().isEmpty()) {
            filteredEvents.retainAll(filterService.findEventsByDateInterval(filterForm.dateStart(), filterForm.dateEnd(),
events));
        }

        // Filtriamo in base alla categoria dell'evento
        if (filterForm.category() != null && filterForm.category().length > 0 && filterForm.category()[0] !=
Categories.EMPTY) {
            filteredEvents.retainAll(filterService.findEventsByCategory(filterForm.category(), events));
        }

        // Ordiniamo gli eventi filtrati in base alla data
        filteredEvents.sort(Comparator.comparing(Event::getDateTime));

        List<ResponseGetEvents> responseGetEvents = new ArrayList<>();
        List<Event> validEvents = new ArrayList<>();

        LocalDateTime currentDateTime = LocalDateTime.now();

        // Verifichiamo la validità degli eventi
        for (Event event : filteredEvents) {
            if (!event.getIsExpired() && !event.getDateTime().isBefore(currentDateTime)) {
                validEvents.add(event);
            } else {
                event.setExpired(true);
                eventService.updateEvent(event);
            }
        }

        // Popoliamo la risposta con gli eventi validi
        for (Event event : validEvents) {
            responseGetEvents.add(new ResponseGetEvents(event.getId(), event.getTitle(),
                event.getCategory().name(), event.getDescription(),
                event.getPlace(), event.getDateTime().toString(),
                "/api/download/" + event.getPhotos().get(0).getId()));
        }

        // Restituiamo una risposta HTTP con la lista di eventi filtrati
        return ResponseEntity.ok(responseGetEvents);
    }
}
```

NotificationController.java

Descrizione: Il file `NotificationController.java` contiene la definizione di un controller Spring Boot che gestisce operazioni legate alle notifiche per gli utenti. Questo controller definisce due endpoint REST che consentono di visualizzare le notifiche e di marcarle come lette.

Caratteristiche principali:

Classe Controller: La classe `NotificationController` è annotata con `@RestController`, il che la identifica come una classe di controller Spring che gestisce le richieste HTTP.
Dependency Injection: La classe fa uso di dependency injection per iniettare una dipendenza di tipo `NotificationService` nel costruttore. Questo è reso possibile dall'annotazione `@RequiredArgsConstructor`, che crea un costruttore che richiede un'istanza di `NotificationService`.
Endpoint `GET /show-notification/{userId}`:

- Questo endpoint è mappato all'URL `/show-notification/{userId}` e viene utilizzato per ottenere le notifiche di un utente specifico. Il parametro `{userId}` è una variabile di percorso che rappresenta l'ID dell'utente.
- Il metodo `ShowNotifications` ottiene la lista delle notifiche per l'utente specificato, crea oggetti `NotificationForm` per rappresentare le notifiche, e restituisce una risposta HTTP contenente queste notifiche. Se non ci sono notifiche, viene restituita una risposta 404 "Not Found".

Endpoint `POST /setNotificationRead/{userId}`:

- Questo endpoint è mappato all'URL `/setNotificationRead/{userId}` e viene utilizzato per contrassegnare le notifiche di un utente specifico come lette. Il parametro `{userId}` è una variabile di percorso che rappresenta l'ID dell'utente.
- Il metodo `SetNotificationsAsRead` ottiene la lista delle notifiche per l'utente specificato, verifica se ciascuna notifica è già stata contrassegnata come letta, e in caso contrario, la contrassegna come letta utilizzando il metodo `notification.setIsRead(true)` e successivamente chiama `notificationService.updateNotification(notification)`. Infine, restituisce una risposta HTTP con il messaggio "Notifications set correctly" per confermare che le notifiche sono state contrassegnate come lette con successo.

Modello dei Dati: Il controller utilizza i modelli di dati `Notification` e `NotificationForm` per rappresentare le notifiche. Questi modelli contengono informazioni come il messaggio, la data e lo stato di lettura delle notifiche.

Gestione delle Risposte: Il controller utilizza `ResponseEntity` per gestire le risposte HTTP in modo appropriato. In caso di successo, vengono restituite risposte 200 "OK" o 201 "Created", a seconda del contesto. Se non ci sono notifiche da visualizzare, viene restituita una risposta 404 "Not Found".

Codice sorgente:

```
package com.eventify.app.controller.api;

import java.util.ArrayList;
import java.util.List;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RestController;

import com.eventify.app.model.Notification;
import com.eventify.app.model.json.NotificationForm;
import com.eventify.app.service.NotificationService;
import lombok.RequiredArgsConstructor;

@RestController
@RequiredArgsConstructor
public class NotificationController {

    private final NotificationService notificationService;

    @GetMapping("/show-notification/{userId}")
    public ResponseEntity<List<NotificationForm>> ShowNotifications(@PathVariable Long userId) {
        // Otteniamo la lista delle notifiche dell'utente specificato dal parametro "userId"
        List<Notification> notifications = notificationService.showUserNotification(userId);
        // Creiamo una lista di oggetti "NotificationForm" per la risposta
        List<NotificationForm> notificheUser = new ArrayList<>();

        // Verifichiamo se ci sono notifiche disponibili
        if (!notifications.isEmpty()) {
            for (Notification notification : notifications) {
                // Per ogni notifica, creiamo un oggetto "NotificationForm" con i dettagli della notifica
                NotificationForm notificForm = new NotificationForm(notification.getMessage(),
notification.getDateTime(), notification.getIsRead());
                notificheUser.add(notificForm);
            }
            // Restituiamo una risposta HTTP OK con la lista di notifiche dell'utente
            return ResponseEntity.ok(notificheUser);
        } else {
            // Se non ci sono notifiche disponibili, restituiamo una risposta HTTP "not found"
            return ResponseEntity.notFound().build();
        }
    }

    @PostMapping("/setNotificationRead/{userId}")
    public ResponseEntity<String> SetNotificationsAsRead(@PathVariable Long userId) {
        // Otteniamo la lista delle notifiche dell'utente specificato dal parametro "userId"
        List<Notification> notifications = notificationService.getNotificationsByUserId(userId);
```

```
        // Iteriamo attraverso le notifiche
        for (Notification notification: notifications) {
            if (!notification.getIsRead()) {
                // Segniamo le notifiche come lette impostando "isRead" a true
                notification.setIsRead(true);
                // Aggiorniamo la notifica nel servizio
                notificationService.updateNotification(notification);
            }
        }

        // Restituiamo una risposta HTTP OK con un messaggio di conferma
        return ResponseEntity.ok("Notifications set correctly");
    }
}
```

PhotoController.java

Descrizione: Il file `PhotoController.java` contiene la definizione di un controller Spring Boot responsabile della gestione di operazioni legate alle foto, comprese la visualizzazione e il download delle immagini.

Caratteristiche principali:

Classe Controller: La classe `PhotoController` è annotata con `@RestController`, il che la identifica come una classe di controller Spring che gestisce le richieste HTTP.
Dependency Injection: Il controller utilizza l'iniezione di dipendenza con l'annotazione `@Autowired` e `@Qualifier` per iniettare una dipendenza di tipo `IPhotoService` con il nome specifico "mainService" nel campo `photoService`. Ciò permette di utilizzare i servizi associati per gestire le operazioni sulle immagini.
Endpoint `GET /api/photos`:

- Questo endpoint è mappato all'URL `/api/photos` ed è utilizzato per ottenere l'elenco di tutte le foto disponibili.
- Il metodo `getAll` chiama il servizio `photoService` per ottenere tutte le foto e le restituisce come un elenco iterabile (`Iterable<Photo>`).

Endpoint `GET /api/photos/{id}`:

- Questo endpoint è mappato all'URL `/api/photos/{id}` ed è utilizzato per ottenere una foto specifica in base all'ID fornito come parametro.
- Il metodo `getById` chiama il servizio `photoService` per ottenere la foto con l'ID specificato. Se la foto non esiste, viene generata una risposta di errore HTTP 404 "Not Found" utilizzando `ResponseStatusException`. Se la foto esiste, viene restituita.

Endpoint `GET /api/download/{id}`:

- Questo endpoint è mappato all'URL `/api/download/{id}` ed è utilizzato per consentire il download di una foto specifica in base all'ID fornito come parametro.
- Il metodo `DownloadImage` ottiene la foto specificata dal servizio `photoService`. Successivamente, crea una risposta HTTP che consente il download dell'immagine. L'immagine viene restituita come `Resource` ed è associata a un tipo di contenuto specificato (`contentType`) e a un'intestazione (`HttpHeaders.CONTENT_DISPOSITION`) che suggerisce il nome del file durante il download.

Codice sorgente:

```
package com.eventify.app.controller.api;

import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.server.ResponseStatusException;
import org.springframework.core.io.ByteArrayResource;
import org.springframework.core.io.Resource;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;

import com.eventify.app.service.interfaces.IPhotoService;
import com.eventify.app.model.Photo;

@RestController
public class PhotoController {

  @Autowired
  @Qualifier("mainService")
  private IPhotoService photoService;

  public PhotoController() {
    // Costruttore del controller
  }

  @RequestMapping("/api/photos")
  public Iterable<Photo> getAll() {
    // Metodo per ottenere l'elenco di tutte le foto
    return photoService.getAll();
  }

  @RequestMapping("/api/photos/{id}")
  public Photo getById(@PathVariable Long id) {
    // Metodo per ottenere una foto specifica in base all'ID fornito come parametro
```

```
        Optional<Photo> photo = photoService.getById(id);

        if (photo.isEmpty()) {
            // Se la foto non esiste, genera una risposta di errore HTTP 404 "Not Found" utilizzando
ResponseStatusException
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, "item not found");
        }
        return photo.get();
    }

    @GetMapping("/api/download/{id}")
    public ResponseEntity<Resource> DownloadImage(@PathVariable Long id) throws Exception {
        // Metodo per consentire il download di una foto specifica in base all'ID fornito come parametro

        Photo photo = photoService.getById(id).get();
        // Ottiene la foto specificata dal servizio IPhotoService

        return ResponseEntity.created(null)
            .contentType(MediaType.parseMediaType(photo.getPhotoType()))
            .header(HttpHeaders.CONTENT_DISPOSITION,
                    "photo; filename=\"" + photo.getPhotoName() + "\"")
            .body(new ByteArrayResource(photo.getData()));
        // Crea una risposta HTTP per consentire il download dell'immagine. L'immagine è restituita come un'istanza
di Resource,
        // ed è associata a un tipo di contenuto specificato (contentType) e a un'intestazione
(HttpHeaders.CONTENT_DISPOSITION)
        // che suggerisce il nome del file durante il download.
    }
}
```

SseController.java

Il file `SseController` è un controller Spring Boot che gestisce le operazioni per Server-Sent Events (SSE), una tecnologia che consente al server di inviare aggiornamenti agli utenti tramite una connessione HTTP persistente, si utilizza questa tecnica per inviare in tempo reale le notifiche agli utenti. Ecco una descrizione delle caratteristiche principali del codice senza il codice stesso:

1. Controller SSE: La classe `SseController` è annotata con `@RestController` e `@RequestMapping("/teller")`. Questo indica che è un controller REST e tutte le sue route inizieranno con `/teller`.
2. Gestione degli SseEmitter: Questa classe gestisce gli oggetti `SseEmitter`, che rappresentano le connessioni SSE ai client. Quando un client si sottoscrive a un flusso, il server crea un `SseEmitter` per mantenere aperta la connessione e invia aggiornamenti attraverso di essa.

3. Log e registrazione: La classe utilizza SLF4J per la registrazione. Vengono registrati eventi importanti come la creazione di un emettitore e il completamento o il timeout di un emettitore.
4. Metodo `streamSse`: Questo metodo è annotato con `@GetMapping("/subscribe/{subscriberId}")` e viene chiamato quando un client richiede di sottoscriversi a un flusso SSE. Il parametro `{subscriberId}` rappresenta l'ID del sottoscrittore. Il metodo restituisce un oggetto `SseEmitter` che rappresenta la connessione SSE verso il client.
5. Timeout del `SseEmitter`: Viene impostato un timeout sul `SseEmitter` con una durata di 3.600.000 millisecondi (ovvero 1 ora). Se il client non riceve aggiornamenti entro questo intervallo, l'emettitore scade, viene completato e rimosso dal servizio.
6. Aggiunta e rimozione di Emitters dal servizio: L'emettitore viene aggiunto al servizio SSE (presumibilmente `SseService`) in modo che possa essere utilizzato per inviare aggiornamenti a un client specifico. Inoltre, vengono definiti callback per gestire eventi come il timeout e il completamento dell'emettitore. Quando l'emettitore scade o viene completato, viene rimosso dal servizio SSE.
7. Server-Sent Events (SSE): SSE è una tecnologia che consente al server di inviare dati al client su una connessione HTTP aperta. È spesso utilizzata per implementare funzionalità di notifica in tempo reale nei browser web.
8. Client di sottoscrizione: I client che desiderano ricevere aggiornamenti da questo flusso SSE devono fare una richiesta GET a `/teller/subscribe/{subscriberId}` specificando un ID di sottoscrittore univoco. Il server invierà quindi gli aggiornamenti tramite il corrispondente `SseEmitter`.

Codice sorgente:

```
package com.eventify.app.controller.api;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.mvc.method.annotation.SseEmitter;

import com.eventify.app.service.SseService;

@RestController
@RequestMapping("/teller")
public class SseController {
   private final Logger logger = LoggerFactory.getLogger(SseController.class);

   // Metodo per gestire le richieste di sottoscrizione SSE
   @GetMapping("/subscribe/{subscriberId}")
   public SseEmitter streamSse(@PathVariable String subscriberId) {
      // Creazione di un nuovo SseEmitter con un timeout specificato
```

```
        SseEmitter emitter = new SseEmitter(3_600_000L);
        logger.info("Emitter created with timeout {} for subscriberId {}", emitter.getTimeout(), subscriberId);

        // Aggiunta dell'emettitore al servizio SSE
        SseService.addEmitter(subscriberId, emitter);

        // Definizione di un callback in caso di timeout
        emitter.onTimeout(() -> {
            logger.info("Emitter timed out");
            emitter.complete();
            // Rimozione dell'emettitore dal servizio SSE
            SseService.removeEmitter(subscriberId);
        });

        // Definizione di un callback in caso di completamento
        emitter.onCompletion(() -> {
            logger.info("Emitter completed");
            // Rimozione dell'emettitore dal servizio SSE
            SseService.removeEmitter(subscriberId);
        });

        // Restituzione dell'emettitore al client
        return emitter;
    }
}

// Record che rappresenta una semplice risposta (non mostrato nel codice)
record SimpleResponse(String content) {
}
```

UserController.java

Il file `UserController` gestisce varie operazioni legate agli utenti e alle modifiche dei loro profili. Ecco una documentazione tecnica con spiegazioni dettagliate delle caratteristiche:

Controller Mapping: Questo controller è annotato con `@RestController` e definisce il percorso radice delle richieste relative agli utenti con `@RequestMapping("/api")`.
Dipendenze Iniettate: Il controller utilizza diverse dipendenze iniettate tramite il costruttore, inclusi servizi come `UserService`, `EmailService`, `PhotoService`, `NotificationService`, `UserValidator`, e `AuthService`.
Conferma Cambio Email:

- Il metodo `confirmEmailChange` è annotato con `@PutMapping("/api/confirm-email")` e gestisce la conferma del cambio dell'email da parte dell'utente.
- Verifica l'OTP (One-Time Password) fornito dall'utente e confronta con quello salvato per l'utente.

- Consente di quindi cambiare le informazioni dell’user in caso ci sia stato anche un cambio email dove a questo punto bisogna effettuare una conferma a due fattori
- Invia una notifica di aggiornamento del profilo e un'email di conferma di cambio email.
- Restituisce una risposta appropriata in base all'esito dell'operazione.

Recupero Informazioni Profilo:

- Il metodo `getProfileInfo` è annotato con `@GetMapping("/api/getProfileInfo/{userId}")` e recupera le informazioni di profilo per un utente specifico.
- Restituisce un oggetto `ResponseProfileInfo` che contiene informazioni come data di nascita, email, nome, cognome e URL dell'immagine del profilo.

Modifica Profilo (Con Caricamento Immagine):

- Il metodo `modifyProfileMultipart` è annotato con `@PutMapping("/api/modify-profile/{userId}")` e gestisce la modifica del profilo dell'utente.
- Controlla e convalida i nuovi dati forniti dall'utente, inclusi il nome, il cognome e l'immagine del profilo.
- Gestisce le modifiche e invia notifiche in caso di successo.
- Supporta l'upload di una nuova immagine del profilo come file multipart.

Modifica Profilo (Senza Caricamento Immagine):

- Il metodo `modifyProfile` è simile a `modifyProfileMultipart` ma non supporta il caricamento di nuove immagini del profilo. Gestisce la modifica del nome, cognome ed email dell'utente.

Validazione dei Dati Utente:

- Utilizza il servizio `UserValidator` per convalidare i dati dell'utente come nome, cognome ed email, assicurandosi che rispettino i criteri di validità.

Questo controller è responsabile della gestione delle operazioni relative agli utenti e alla modifica dei loro profili, inclusa la convalida dei dati e l'invio di notifiche.
Codice sorgente:

```
package com.eventify.app.controller.api;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Optional;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
```

```
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.multipart.MultipartFile;

import com.eventify.app.model.Photo;
import com.eventify.app.model.User;
import com.eventify.app.model.json.ResponseProfileInfo;
import com.eventify.app.service.AuthService;
import com.eventify.app.service.EmailService;
import com.eventify.app.service.NotificationService;
import com.eventify.app.service.PhotoService;
import com.eventify.app.service.UserService;
import com.eventify.app.validator.UserValidator;

import jakarta.mail.MessagingException;
import lombok.AllArgsConstructor;

@RestController
@AllArgsConstructor
public class UserController {

   private final UserService userService;
   private final EmailService emailService;
   private final PhotoService photoService;
   private final NotificationService notificationService;
   private final UserValidator userValidator;
   private final AuthService authService;

   // Conferma il cambio di email dell'utente
   @PutMapping("/api/confirm-email")
   public ResponseEntity<String> confirmEmailChange(@PathVariable Long userId, @RequestParam String
firstname, @RequestParam String lastname, @RequestParam String email, @RequestParam String birth,
@RequestParam MultipartFile pics, @RequestParam String otpCode) throws Exception {
      Optional<User> userOptional = userService.getById(userId);
      int otp = Integer.parseInt(otpCode);
      User user = userOptional.get();

      if (user.getOtp() == otp) {
         if (firstname != null)
            user.setFirstname(firstname);
         if (lastname != null)
            user.setLastname(lastname);
         if (birth != null) {
            SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
            try {
               Date date = formatter.parse(birth);
               user.setDob(date);
            } catch (ParseException e) {
               e.printStackTrace();
               return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore nel formattare la
data di nascita.");
            }
         }
         if (pics != null) {
            try {
```

```
                Photo profilePics = photoService.uploadPhoto(pics);
                user.setProfilePicture(profilePics);
            } catch (MessagingException e) {
                e.printStackTrace();
            }
        }
        userService.update(userId, user);
        try {
            emailService.sendChangesAdviseAboutProfile(email);
            notificationService.createNotification(userId, null, "I tuoi dati sono stati aggiornati.", null);
        } catch (MessagingException e) {
            e.printStackTrace();
            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore nell'invio dell'email.");
        }
        return ResponseEntity.ok("Profilo aggiornato con successo.");
    }
    return ResponseEntity.ok("Codice OTP errato.");
}

// Ottiene le informazioni del profilo dell'utente
@GetMapping("/api/getProfileInfo/{userId}")
public ResponseEntity<ResponseProfileInfo> getProfileInfo(@PathVariable Long userId) {
    Optional<User> user = userService.getById(userId);

    if (user.isEmpty()) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(null);
    }

    return ResponseEntity.ok(ResponseProfileInfo.builder()
            .date(user.get().getDob().toString())
            .email(user.get().getEmail())
            .firstName(user.get().getFirstname())
            .lastName(user.get().getLastname())
            .imageUrl("/api/download/" + user.get().getProfilePicture().getId())
            .build());
}

// Modifica il profilo con il caricamento di un'immagine del profilo
@PutMapping("/api/modify-profile/{userId}")
public ResponseEntity<String> modifyProfileMultipart(@PathVariable Long userId,
@RequestParam("firstname") String firstname, @RequestParam("lastname") String lastname,
@RequestParam("email") String email, @RequestParam("profilePhoto") MultipartFile pics) throws Exception {
    Optional<User> userOptional = userService.getById(userId);

    if (userOptional.isPresent()) {
        if (firstname.length() != 0) {
            if (!userValidator.isValidName(firstname))
                return ResponseEntity.ok("Nome dell'utente non valido.");
        }
        if (lastname.length() != 0) {
            if (!userValidator.isValidName(lastname))
                return ResponseEntity.ok("Cognome dell'utente non valido.");
        }
        if (pics != null) {
            if (!userValidator.isImageFile(pics))
                return ResponseEntity.ok("Immagine non in formato PNG, JPG o JPEG");
```

```
            }
            if (email.length() == 0) {
                User user = userOptional.get();
                if (firstname.length() != 0)
                    user.setFirstname(firstname);
                if (lastname.length() != 0)
                    user.setLastname(lastname);
                if (pics != null) {
                    try {
                        Photo profilePics = photoService.uploadPhoto(pics);
                        photoService.create(profilePics);
                        user.setProfilePicture(profilePics);
                    } catch (Exception e) {
                        e.getMessage();
                    }
                }
                userService.update(userId, user);
                try {
                    emailService.sendChangesAdviseAboutProfile(user.getEmail());
                    notificationService.createNotification(userId, null, "Il tuo profilo è stato aggiornato", null);
                } catch (MessagingException e) {
                    e.printStackTrace();
                    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore nell'invio
dell'email.");
                }
                return ResponseEntity.ok("Profilo aggiornato con successo.");
            } else {
                if (!userValidator.isValidEmail(email))
                    return ResponseEntity.ok("L'indirizzo email non è valido.");
                String secretKey = authService.generateSecretKey();
                int otp = authService.generateOtp(secretKey);
                User user = userOptional.get();
                user.setOtp(otp);
                try {
                    emailService.sendConfirmChangeEmail(email, otp);
                } catch (MessagingException e) {
                    e.printStackTrace();
                    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore nell'invio
dell'email.");
                }
                return ResponseEntity.ok("L'email non è stata ancora convalidata.");
            }
        }
        return ResponseEntity.ok("Utente non trovato.");
    }

    // Modifica il profilo senza caricamento di immagine del profilo
    @PutMapping("/api/modify-profile-multipart/{userId}")
    public ResponseEntity<String> modifyProfile(@PathVariable Long userId, @RequestParam("firstname") String
firstname, @RequestParam("lastname") String lastname, @RequestParam("email") String email) throws
Exception {
        Optional<User> userOptional = userService.getById(userId);

        if (userOptional.isPresent()) {
            if (firstname.length() != 0) {
                if (!userValidator.isValidName(firstname))
```

```
                return ResponseEntity.ok("Nome dell'utente non valido.");
            }
            if (lastname.length() != 0) {
                if (!userValidator.isValidName(lastname))
                    return ResponseEntity.ok("Cognome dell'utente non valido.");
            }
            if (email.length() == 0) {
                User user = userOptional.get();
                if (firstname.length() != 0)
                    user.setFirstname(firstname);
                if (lastname.length() != 0)
                    user.setLastname(lastname);

                    userService.update(userId, user);
                try {
                    emailService.sendChangesAdviseAboutProfile(user.getEmail());
                    notificationService.createNotification(userId, null, "Il tuo profilo è stato aggiornato", null);
                } catch (MessagingException e) {
                    e.printStackTrace();
                    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore nell'invio
dell'email.");
                }
                return ResponseEntity.ok("Profilo aggiornato con successo.");
            } else {
                if (!userValidator.isValidEmail(email))
                    return ResponseEntity.ok("L'indirizzo email non è valido.");
                String secretKey = authService.generateSecretKey();
                int otp = authService.generateOtp(secretKey);
                User user = userOptional.get();
                user.setOtp(otp);
                try {
                    emailService.sendConfirmChangeEmail(email, otp);
                } catch (MessagingException e) {
                    e.printStackTrace();
                    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Errore nell'invio
dell'email.");
                }
                return ResponseEntity.ok("L'email non è stata ancora convalidata.");
            }
        }
        return ResponseEntity.ok("Utente non trovato.");
    }
}
```

Modelli

Event.java

Descrizione: Il file "Event.java" contiene la definizione di una classe denominata "Event" che rappresenta un evento all'interno dell'applicazione "eventify". Questa

classe è annotata con diverse annotazioni, come `@Entity` e `@Table`, per mappare la classe agli oggetti del database e definire la struttura della tabella nel database.

Caratteristiche:

- La classe "Event" rappresenta un evento e ha una serie di attributi che definiscono un evento, come "id", "title", "description", "dateTime", "place", ecc.
- Gli attributi della classe sono annotati con diverse annotazioni come `@Id`, `@Column`, `@ManyToOne`, `@OneToMany`, `@ManyToMany`, `@NotNull`, `@Builder.Default` e altre. Queste annotazioni sono utilizzate per definire la mappatura degli attributi con le colonne del database, specificare vincoli come la non nullità e gestire le relazioni tra entità.
- La classe "Event" ha una relazione uno-a-molti con la classe "Photo", in quanto un evento può avere molte foto associate. Questa relazione è gestita dall'attributo "photos" della classe "Event".
- La classe "Event" ha anche una relazione molti-a-molti con la classe "User" per rappresentare i partecipanti all'evento. Questa relazione è gestita dall'attributo "participants" della classe "Event".
- Ogni evento ha un creatore, che è un oggetto di tipo "User". Questa relazione è gestita dall'attributo "creator" della classe "Event".
- La classe "Event" è annotata con `@Table(name = "event", schema = "eventify")`, che specifica il nome della tabella nel database e lo schema in cui si trova.
- L'enumerazione "Categories" definisce le categorie a cui può appartenere un evento, ad esempio "Party", "Sport", "Music", ecc.
- L'attributo "isExpired" è un flag booleano che indica se l'evento è scaduto o meno.

Utilizzo tipico: La classe "Event" è utilizzata per rappresentare gli eventi all'interno dell'applicazione "eventify". Gli oggetti di questa classe vengono creati, letti e modificati dal sistema per gestire gli eventi e i loro dettagli, tra cui titolo, descrizione, data e ora, luogo, partecipanti e creatore. Questi eventi possono essere visualizzati dagli utenti dell'applicazione e possono essere associati a foto.

Codice sorgente:

```
package com.eventify.app.model;

import java.time.LocalDateTime;
import java.util.*;

import com.eventify.app.model.enums.Categories;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
```

```
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.JoinTable;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.OneToMany;
import jakarta.persistence.Table;
import jakarta.validation.constraints.NotNull;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "event", schema = "eventify")
public class Event {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    @Column(nullable = false)
    private String title;

    @NotNull
    @Column(nullable = false)
    private String description;

    @NotNull
    @Column(nullable = false)
    private LocalDateTime dateTime;

    @NotNull
    @Column(nullable = false)
    private String place;

    @OneToMany(mappedBy = "event", fetch = FetchType.EAGER)
    private List<Photo> photos;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
        name = "event_participants",
        joinColumns = @JoinColumn(name = "event_id"),
        inverseJoinColumns = @JoinColumn(name = "user_id")
    )
    @Builder.Default
    private List<User> participants = new ArrayList<>();;

    @ManyToOne
    @JoinColumn(name = "creator_id", referencedColumnName = "id")
    private User creator;
```

```
    @NotNull
    private Categories category;

    @NotNull
    private boolean isExpired;

    public Event(String title, String description, LocalDateTime dateTime, String place, List<Photo> photos,
List<User> participants, User creator, Categories category) {
        this.title = title;
        this.description = description;
        this.dateTime = dateTime;
        this.place = place;
        this.photos = photos;
        this.participants = participants;
        this.creator = creator;
        this.category = category;
    }

    public Long getId() {
        return id;
    }

    public boolean getIsExpired() {
        return this.isExpired;
    }

    public void setExpired(boolean expired) {
        this.isExpired = expired;
    }

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

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public LocalDateTime getDateTime() {
        return dateTime;
    }

    public void setDateTime(LocalDateTime dateTime) {
        this.dateTime = dateTime;
    }
```

```
    public String getPlace() {
        return place;
    }

    public void setPlace(String place) {
        this.place = place;
    }

    public List<Photo> getPhotos() {
        return this.photos;
    }

    public void setPhotos(List<Photo> photos) {
        this.photos = photos;
    }

    public Categories getCategory() {
        return category;
    }

    public void setCategory(Categories category) {
        this.category = category;
    }

    public List<User> getParticipants() {
        return participants;
    }

    public void setParticipants(List<User> participants) {
        this.participants = participants;
    }

    public User getCreator() {
        return creator;
    }

    public void setCreator(User creator) {
        this.creator = creator;
    }

    @Override
    public String toString() {
        return "YourClass{" +
                "id=" + id +
                ", title='" + title + '\'' +
                ", description='" + description + '\'' +
                ", dateTime=" + dateTime +
                ", place='" + place + '\'' +
                ", category=" + category +
                '}';
    }
}
```

User,java

Descrizione: Il file "User.java" contiene la definizione di una classe denominata "User" che rappresenta un utente all'interno dell'applicazione "eventify". Questa classe è annotata con diverse annotazioni, come `@Entity` e `@Table`, per mappare la classe agli oggetti del database e definire la struttura della tabella nel database. Inoltre, implementa l'interfaccia "UserDetails" per consentire l'integrazione con Spring Security.

Caratteristiche:

- La classe "User" rappresenta un utente dell'applicazione ed è utilizzata per memorizzare informazioni sull'utente, come nome, cognome, data di nascita, email, password, ruolo, immagine del profilo, token di aggiornamento e altro.
- Gli attributi della classe sono annotati con diverse annotazioni come `@Id`, `@Column`, `@ManyToOne`, `@OneToMany`, `@ManyToMany`, `@NotNull`, `@Size`, `@Email`, `@Temporal`, `@DateTimeFormat`, `@Pattern`, `@Enumerated` e altre. Queste annotazioni sono utilizzate per definire la mappatura degli attributi con le colonne del database, specificare vincoli come la non nullità e il formato dei dati, e gestire le relazioni tra entità.
- La classe "User" implementa l'interfaccia "UserDetails" di Spring Security, che consente di utilizzare oggetti "User" come oggetti UserDetails per l'autenticazione e l'autorizzazione.
- Gli utenti possono avere un ruolo specifico, che è un'enumerazione di tipo "Role" (come "USER" o "ADMIN").
- Gli utenti possono anche avere un token di aggiornamento e un codice OTP (One-Time Password) per scopi di autenticazione e sicurezza.
- Gli utenti possono essere creatori di eventi (relazione uno-a-molti) e partecipare a eventi (relazione molti-a-molti), gestite rispettivamente dagli attributi "createdEvents" e "events".
- La classe "User" è annotata con `@Table(name = "user", schema = "eventify")`, che specifica il nome della tabella nel database e lo schema in cui si trova.

Utilizzo tipico: La classe "User" è utilizzata per rappresentare gli utenti registrati all'interno dell'applicazione "eventify". Gli oggetti di questa classe vengono creati durante la registrazione degli utenti e memorizzati nel database. Questi oggetti vengono utilizzati per gestire l'identità degli utenti, inclusi i dettagli personali, le informazioni sull'autenticazione e i ruoli. La classe consente inoltre la gestione delle relazioni tra utenti ed eventi, compresa la creazione di eventi da parte degli utenti e la partecipazione agli eventi.

Codice sorgente:

```
package com.eventify.app.model;

import java.util.Collection;
import java.util.Date;
import java.util.List;

import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

import com.eventify.app.model.enums.Role;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.OneToMany;
import jakarta.persistence.Table;

import jakarta.persistence.Temporal;
import jakarta.persistence.TemporalType;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Pattern;
import jakarta.validation.constraints.Size;

import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Entity
@NoArgsConstructor
@Table(name = "user", schema = "eventify")
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    @Column(nullable = false)
    private String firstname;

    @NotNull
    @Column(nullable = false)
    private String lastname;

    @NotNull
    @Temporal(TemporalType.DATE)
    @DateTimeFormat(pattern = "yyyy-MM-dd")
```

```
private Date dob;

@Email
@NotNull
@Column(nullable = false, unique = true)
@Pattern(regexp = "^[A-Za-z0-9+_.-]+@([A-Za-z0-9.-]+)\\.[A-Za-z]{2,4}$")
private String email;

@NotNull
@Size(min = 8, max = 100)
@Column(nullable = false)
private String password;

@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "profile_picture", referencedColumnName = "id")
private Photo profilePicture;

@Enumerated(EnumType.STRING)
private Role role;

public String refreshToken;

public Integer otp;

@OneToMany(mappedBy = "creator")
private List<Event> createdEvents;

@ManyToMany(mappedBy = "participants")
private List<Event> events;

public User(String firstname, String lastname, Date dob, String email, String password, Photo profilePicture) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.dob = dob;
    this.email = email;
    this.password = password;
    this.profilePicture = profilePicture;
}

public Long getId() {
    return id;
}

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

public String getFirstname() {
    return this.firstname;
}

public void setFirstname(String firstname) {
    this.firstname = firstname;
}

public String getLastname() {
    return this.lastname;
}
```

```
public void setLastname(String lastname) {
    this.lastname = lastname;
}

public Date getDob() {
    return this.dob;
}

public void setDob(Date dob) {
    this.dob = dob;
}

public String getEmail() {
    return this.email;
}

public void setEmail(String email) {
    this.email = email;
}

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

public Photo getProfilePicture() {
    return this.profilePicture;
}

public void setProfilePicture(Photo profilePicture) {
    this.profilePicture = profilePicture;
}

public Role getRole() {
    return this.role;
}

public void setRole(Role role) {
    this.role = role;
}

public String getRefreshToken() {
    return this.refreshToken;
}

public void setRefreshToken(String refreshToken) {
    this.refreshToken = refreshToken;
}

public Integer getOtp() {
    return this.otp;
}

public void setOtp(Integer otp) {
    this.otp = otp;
}

@Override
```

```
    public String toString() {
        return "Eventi{" +
                "id=" + id +
                ", firstname='" + firstname + '\'' +
                ", lastname=" + lastname +
                ", dob=" + dob +
                ", email='" + email + '\'' +
                ", password='" + password + '\'' +
                ", profilePicture='" + profilePicture + '\'' +
                ", role='" + role + '\'' +
                '}';
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new SimpleGrantedAuthority(role.name()));
    }

    @Override
    public String getUsername() {
        return this.email;
    }

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

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

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

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

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

Notification.java

Descrizione: Il file "Notification.java" contiene la definizione della classe "Notification", che rappresenta una notifica all'interno dell'applicazione "eventify".

Questa classe è utilizzata per memorizzare le informazioni relative alle notifiche degli utenti e consente loro di ricevere messaggi o avvisi all'interno dell'applicazione.

Caratteristiche:

- La classe "Notification" rappresenta una notifica ed è annotata con diverse annotazioni, tra cui `@Entity` e `@Table`, che sono utilizzate per mappare la classe agli oggetti del database e definire la struttura della tabella nel database.
- La classe contiene vari attributi, tra cui:
    - `id`: un identificatore univoco per ogni notifica.
    - `user`: un riferimento all'utente a cui è destinata la notifica. La relazione tra notifica e utente è gestita dalla relazione molti-a-uno specificata dall'annotazione `@ManyToOne`.
    - `message`: il messaggio o il contenuto della notifica.
    - `dateTime`: la data e l'ora di creazione della notifica.
    - `isRead`: un flag booleano che indica se la notifica è stata letta o meno.
- La classe dispone di un costruttore che accetta un utente e un messaggio come argomenti e inizializza gli attributi corrispondenti con i valori forniti. La data e l'ora di creazione vengono impostate sulla data corrente, e il flag "isRead" viene inizializzato a "false".
- Gli attributi della classe sono definiti con i relativi metodi getter e setter.
- La classe è annotata con `@Table(name = "notification", schema = "eventify")`, specificando il nome della tabella nel database e lo schema in cui si trova.

Utilizzo tipico: La classe "Notification" è utilizzata per memorizzare e gestire le notifiche destinate agli utenti all'interno dell'applicazione "eventify". Le notifiche vengono create e memorizzate nel database quando è necessario inviare messaggi, avvisi o comunicazioni agli utenti. Gli utenti possono accedere alle loro notifiche e marcarle come lette. La classe "Notification" consente di tenere traccia delle notifiche create, dei destinatari, del contenuto del messaggio e dello stato di lettura.

Codice sorgente:

```
package com.eventify.app.model;

import java.util.Date;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
```

```
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Entity
@NoArgsConstructor
@Table(name = "notification", schema = "eventify")
public class Notification {

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

    @ManyToOne
    @JoinColumn(name = "User_id")
    private User user;

    private String message;

    @Column(name = "data_creazione", nullable = false)
    private String dateTime;

    @Column(name = "is_read", nullable = false)
    private boolean isRead;

    public Notification(User user, String message) {
        this.user = user;
        this.message = message;
        this.dateTime = new Date().toString();
        this.isRead = false;
    }

    public Long getId() {
        return this.id;
    }

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

    public User getUser() {
        return this.user;
    }

    public void setUser(User user) {
        this.user = user;
    }

    public String getMessage() {
        return this.message;
    }

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

```
    public String getDateTime() {
        return this.dateTime;
    }

    public void setDateTime(String dateTime) {
        this.dateTime = dateTime;
    }

    public boolean getIsRead() {
        return isRead;
    }

    public void setIsRead(boolean isRead) {
        this.isRead = isRead;
    }

}
```

Photo.java

Descrizione: Il file "Notification.java" contiene la definizione della classe "Notification", che rappresenta una notifica all'interno dell'applicazione "eventify". Questa classe è utilizzata per memorizzare le informazioni relative alle notifiche degli utenti e consente loro di ricevere messaggi o avvisi all'interno dell'applicazione.

Caratteristiche:

- La classe "Notification" rappresenta una notifica ed è annotata con diverse annotazioni, tra cui `@Entity` e `@Table`, che sono utilizzate per mappare la classe agli oggetti del database e definire la struttura della tabella nel database.
- La classe contiene vari attributi, tra cui:
    - `id`: un identificatore univoco per ogni notifica.
    - `user`: un riferimento all'utente a cui è destinata la notifica. La relazione tra notifica e utente è gestita dalla relazione molti-a-uno specificata dall'annotazione `@ManyToOne`.
    - `message`: il messaggio o il contenuto della notifica.
    - `dateTime`: la data e l'ora di creazione della notifica.
    - `isRead`: un flag booleano che indica se la notifica è stata letta o meno.
- La classe dispone di un costruttore che accetta un utente e un messaggio come argomenti e inizializza gli attributi corrispondenti con i valori forniti. La data e l'ora di creazione vengono impostate sulla data corrente, e il flag "isRead" viene inizializzato a "false".
- Gli attributi della classe sono definiti con i relativi metodi getter e setter.

- La classe è annotata con `@Table(name = "notification", schema = "eventify")`, specificando il nome della tabella nel database e lo schema in cui si trova.

Utilizzo tipico: La classe "Notification" è utilizzata per memorizzare e gestire le notifiche destinate agli utenti all'interno dell'applicazione "eventify". Le notifiche vengono create e memorizzate nel database quando è necessario inviare messaggi, avvisi o comunicazioni agli utenti. Gli utenti possono accedere alle loro notifiche e marcarle come lette. La classe "Notification" consente di tenere traccia delle notifiche create, dei destinatari, del contenuto del messaggio e dello stato di lettura.

Codice sorgente:

```
package com.eventify.app.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.Lob;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;
import jakarta.validation.constraints.NotNull;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@Entity
@Table(name = "photo", schema = "eventify")
public class Photo {

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

	@NotNull
	private String photoName;

	@NotNull
	private String photoType;

	@NotNull
	private Boolean is_deleted;

	@Lob
	@NotNull
	private byte[] data;

	@ManyToOne
	@JoinColumn(name = "event_id")
	private Event event;
```

```
	public Photo(String photoType, String photoName, Boolean is_deleted, byte[] data) {
		this.photoType = photoType;
		this.photoName = photoName;
		this.is_deleted = is_deleted;
		this.data = data;
	}

	public Long getId() {
		return (id);
	}

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

	public String getPhotoType() {
		return (this.photoType);
	}

	public void setPhotoType(String photoType) {
		this.photoType = photoType;
	}

	public String getPhotoName() {
		return (this.photoName);
	}

	public void setPhotoName(String photoName) {
		this.photoName = photoName;
	}

	public Boolean getIsDeleted() {
		return (this.is_deleted);
	}

	public void setIsDeleted(Boolean is_deleted) {
		this.is_deleted = is_deleted;
	}

	public byte[] getData() {
		return (this.data);
	}

	public void setData(byte[] data) {
		this.data = data;
	}

    @Override
    public String toString() {
        return "Eventi{" +
                "id=" + id +
                ", photoName='" + photoName + '\'' +
                ", photoType=" + photoType +
                ", data=" + data +
                ", is_deleted='" + is_deleted + '\'' +
                '}';
    }
}
```

Repository

IEventRepository.java

Descrizione: Il file "IEventRepository.java" contiene l'interfaccia "IEventRepository", che definisce i metodi di accesso ai dati per la gestione delle entità "Event" nel contesto dell'applicazione "eventify". Questa interfaccia è utilizzata per eseguire operazioni di lettura e scrittura sui dati relativi agli eventi memorizzati nel database.

Caratteristiche:

- L'interfaccia "IEventRepository" estende l'interfaccia "JpaRepository<Event, Long>", che è fornita da Spring Data JPA. Ciò significa che questa interfaccia eredita tutti i metodi di base di un repository JPA (ad esempio, metodi per il recupero, la creazione, l'aggiornamento e l'eliminazione di entità).
- L'interfaccia "IEventRepository" dichiara vari metodi personalizzati per l'accesso ai dati relativi agli eventi, come ad esempio:
    - `List<Event> findByTitleContaining(String keyword)`: Questo metodo consente di cercare eventi il cui titolo contiene una determinata parola chiave.
    - `Optional<Event> findByTitle(String title)`: Questo metodo consente di cercare un evento specifico in base al titolo.
    - `List<Event> findByPlace(String place)`: Questo metodo consente di cercare eventi in base al luogo in cui si svolgono.
    - `List<Event> findByCategory(Categories category)`: Questo metodo consente di cercare eventi in base alla categoria a cui appartengono.
    - `List<Event> findByCreator(User creator)`: Questo metodo consente di cercare eventi creati da un utente specifico.
    - `List<Event> findByParticipantsContaining(User user)`: Questo metodo consente di cercare eventi a cui partecipa un utente specifico.
- Gli oggetti restituiti da questi metodi sono rappresentati come liste di eventi o oggetti opzionali di tipo "Event".
- Questi metodi personalizzati utilizzano le convenzioni di denominazione di Spring Data JPA per generare automaticamente le query SQL necessarie per eseguire le ricerche. Ad esempio, il metodo `findByTitleContaining` genererà una query SQL per cercare eventi con titoli contenenti la parola chiave specificata.
- L'interfaccia non contiene alcuna implementazione dei metodi, poiché Spring Data JPA si occuperà di generare le implementazioni dei repository in fase di esecuzione.

Utilizzo tipico

L'interfaccia "IEventRepository" è utilizzata come un repository per accedere ai dati relativi agli eventi all'interno dell'applicazione "eventify". Gli sviluppatori utilizzano questa interfaccia per eseguire operazioni di lettura e scrittura sui dati degli eventi, come la ricerca di eventi in base a criteri specifici (titolo, luogo, categoria, creatore, partecipanti) e il recupero di informazioni sugli eventi dal database. La gestione dei dati degli eventi è fondamentale per l'applicazione, poiché consente agli utenti di visualizzare, partecipare e creare eventi.
Codice sorgente:

```
package com.eventify.app.repository;

import java.util.List;
import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;

import com.eventify.app.model.Event;
import com.eventify.app.model.User;
import com.eventify.app.model.enums.Categories;

public interface IEventRepository extends JpaRepository<Event, Long> {
    // Metodo per cercare eventi il cui titolo contiene una parola chiave
    List<Event> findByTitleContaining(String keyword);

    // Metodo per cercare un evento specifico in base al titolo
    Optional<Event> findByTitle(String title);

    // Metodo per cercare eventi in base al luogo in cui si svolgono
    List<Event> findByPlace(String place);

    // Metodo per cercare eventi in base alla categoria a cui appartengono
    List<Event> findByCategory(Categories category);

    // Metodo per cercare eventi creati da un utente specifico
    List<Event> findByCreator(User creator);

    // Metodo per cercare eventi a cui partecipa un utente specifico
    List<Event> findByParticipantsContaining(User user);
}
```

IUserRepository.java

Descrizione:

Questo file rappresenta un'interfaccia Java denominata "IUserRepository" che fa parte del repository del framework Spring Data JPA. Spring Data JPA semplifica l'accesso ai dati dal database relazionale e la scrittura di query SQL personalizzate, offrendo una serie di funzionalità pronte all'uso.

Caratteristiche:

- L'interfaccia estende "JpaRepository<User, Long>". Questo indica che "IUserRepository" gestisce oggetti di tipo "User" e utilizza identificatori di tipo "Long" per rappresentare le chiavi primarie.
- L'annotazione `@Repository` è utilizzata per dichiarare questa interfaccia come un componente gestito da Spring, consentendo a Spring di gestire l'iniezione delle dipendenze e fornire implementazioni per i metodi definiti nell'interfaccia.
- Questa interfaccia non contiene metodi personalizzati dichiarati. Tuttavia, eredita una serie di metodi generici forniti da "JpaRepository" per eseguire operazioni CRUD (Create, Read, Update, Delete) sugli oggetti "User" nel database.
- I metodi generici ereditati includono il salvataggio di un utente, la ricerca di un utente per ID, la cancellazione di un utente e altre operazioni comuni.
- Spring Data JPA automaticamente genera implementazioni dei metodi basati sul nome del metodo e sulla firma del metodo dichiarato nell'interfaccia. Ad esempio, se desideri cercare un utente per nome, puoi definire un metodo "findByFirstName" in questa interfaccia, e Spring Data JPA genererà automaticamente una query SQL basata su questo metodo.
- Questa interfaccia rappresenta uno strato di accesso ai dati che consente di interagire con l'entità "User" nel database utilizzando un'interfaccia orientata agli oggetti anziché scrivere manualmente query SQL.

Codice sorgente:

```
package com.eventify.app.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.eventify.app.model.User;

@Repository
public interface IUserRepository extends JpaRepository<User, Long> {
}
```

IPhotoRepository.java

Descrizione:

Questo file rappresenta un'interfaccia Java denominata "IPhotoRepository" che fa parte del repository del framework Spring Data JPA. Spring Data JPA semplifica l'accesso ai dati dal database relazionale e la scrittura di query SQL personalizzate, offrendo una serie di funzionalità pronte all'uso.

Caratteristiche:

- L'interfaccia estende "JpaRepository<Photo, Long>". Questo indica che "IPhotoRepository" gestisce oggetti di tipo "Photo" e utilizza identificatori di tipo "Long" per rappresentare le chiavi primarie.
- Questa interfaccia non dichiara alcun metodo specifico. Tuttavia, eredita tutti i metodi predefiniti da "JpaRepository". Questi metodi predefiniti consentono di effettuare operazioni comuni sulle entità "Photo", ad esempio, inserimento, aggiornamento, eliminazione, ricerca per ID, ricerca di tutti gli oggetti e così via.
- L'annotazione `@Repository` indica a Spring che questa interfaccia è un componente di tipo repository e dovrebbe essere considerata una candidata per la scansione dei componenti. Questo significa che verrà gestita da Spring come una classe di repository e sarà possibile iniettarla in altre parti dell'applicazione per l'accesso ai dati relativi alle foto.

Codice sorgente:

```
package com.eventify.app.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.eventify.app.model.Photo;

@Repository
public interface IPhotoRepository extends JpaRepository<Photo, Long> {
}
```

INotification.java

Descrizione:

Questo file rappresenta un'interfaccia Java denominata "INotificationRepository" che fa parte del repository del framework Spring Data JPA. Spring Data JPA semplifica l'accesso ai dati dal database relazionale e la scrittura di query SQL personalizzate, offrendo una serie di funzionalità pronte all'uso.

Caratteristiche:

- L'interfaccia estende "JpaRepository<Notification, Long>". Questo indica che "INotificationRepository" gestisce oggetti di tipo "Notification" e utilizza identificatori di tipo "Long" per rappresentare le chiavi primarie.

- L'interfaccia dichiara un metodo specifico chiamato "getNotificationsByUser_Id" che è utilizzato per recuperare le notifiche associate a un determinato utente. Questo metodo restituirà una lista di oggetti "Notification" in base all'ID dell'utente specificato come argomento.
- L'annotazione `@Repository` indica a Spring che questa interfaccia è un componente di tipo repository e dovrebbe essere considerata una candidata per la scansione dei componenti. Questo significa che verrà gestita da Spring come una classe di repository e sarà possibile iniettarla in altre parti dell'applicazione per l'accesso ai dati relativi alle notifiche.

Codice sorgente:

```
package com.eventify.app.repository;

import com.eventify.app.model.Notification;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface INotificationRepository extends JpaRepository<Notification, Long> {
   List<Notification> getNotificationsByUser_Id(Long userId);
}
```

Servizi

UserService.java

Descrizione:

Questo file rappresenta un servizio denominato "UserService" all'interno di un'applicazione Spring. Questo servizio è responsabile della gestione degli utenti all'interno del sistema, compresa la registrazione, l'aggiornamento, la cancellazione e la ricerca degli utenti. Il servizio fa ampio uso del repository "IUserRepository" per l'interazione con il database relazionale.

Caratteristiche:

- L'annotazione `@Service` dichiara che questa classe è un componente di servizio di Spring e dovrebbe essere gestita dal container di Spring. Questo consente di iniettare il servizio in altre parti dell'applicazione.

- La classe utilizza l'interfaccia "IUserService" per dichiarare i metodi che fornisce, tra cui "getAll", "getById", "create", "update" e "delete". Questi metodi consentono di eseguire operazioni CRUD (Create, Read, Update, Delete) sugli utenti.
- Nel costruttore di classe, vengono iniettati i componenti necessari, in particolare "IUserRepository" per l'accesso ai dati degli utenti e "BCryptPasswordEncoder" per la crittografia delle password degli utenti.
- Il metodo "getAll" restituisce un elenco di tutti gli utenti nel database.
- Il metodo "getById" restituisce un utente in base all'ID specificato.
- Il metodo "create" consente di creare un nuovo utente. Prima di salvare l'utente, la password viene crittografata utilizzando BCrypt per garantire la sicurezza.
- Il metodo "update" consente di aggiornare un utente esistente in base all'ID specificato. Prima di eseguire l'aggiornamento, viene verificata l'esistenza dell'utente.
- Il metodo "delete" consente di eliminare un utente in base all'ID specificato. Prima di eseguire l'eliminazione, viene verificata l'esistenza dell'utente.
- I metodi "findByEmail" e "findByOtp" consentono di cercare un utente in base all'indirizzo email o al codice OTP specificato. Questi metodi scansionano l'elenco di tutti gli utenti per trovare la corrispondenza.

Codice sorgente:

```
package com.eventify.app.service;

import java.util.List;
import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.stereotype.Service;

import com.eventify.app.service.interfaces.IUserService;
import com.eventify.app.model.User;
import com.eventify.app.repository.IUserRepository;

@Service
public class UserService implements IUserService {

    // Iniezione del repository IUserRepository
    @Autowired
    private IUserRepository userRepository;

    // Costruttore del servizio
    public UserService() {
        // Costruttore vuoto
    }

    // Restituisce un elenco di tutti gli utenti
    @Override
```

```
public Iterable<User> getAll() {
    return userRepository.findAll();
}

// Restituisce un utente in base all'ID specificato
@Override
public Optional<User> getById(Long id) {
    return userRepository.findById(id);
}

// Crea un nuovo utente, crittografando la password prima del salvataggio
@Override
public User create(User user) {
    BCryptPasswordEncoder bcrypt = new BCryptPasswordEncoder();
    String encryptedPwd = bcrypt.encode(user.getPassword());
    user.setPassword(encryptedPwd);
    return userRepository.save(user);
}

// Aggiorna un utente esistente in base all'ID specificato
@Override
public Optional<User> update(Long id, User user) {
    Optional<User> foundUser = userRepository.findById(id);

    if (foundUser.isEmpty()) {
        return Optional.empty();
    }

    foundUser.get().setFirstname(user.getFirstname());
    foundUser.get().setLastname(user.getLastname());
    foundUser.get().setDob(user.getDob());
    foundUser.get().setEmail(user.getEmail());
    foundUser.get().setPassword(user.getPassword());
    foundUser.get().setProfilePicture(user.getProfilePicture());
    userRepository.save(foundUser.get());

    return foundUser;
}

// Elimina un utente in base all'ID specificato
@Override
public Boolean delete(Long id) {
    Optional<User> foundUser = userRepository.findById(id);

    if (foundUser.isEmpty()) {
        return false;
    }

    userRepository.delete(foundUser.get());

    return true;
}

// Cerca un utente in base all'indirizzo email specificato
public Optional<User> findByEmail(String email) {
    List<User> allUsers = userRepository.findAll();
```

```
        for (User user : allUsers) {
            if (user.getEmail().equals(email)) {
                return Optional.of(user);
            }
        }
        return Optional.empty();
    }

    // Cerca un utente in base al codice OTP specificato
    public Optional<User> findByOtp(Integer otp) {
        List<User> allUsers = userRepository.findAll();

        for (User user : allUsers) {
            if (user.getOtp() != null && user.getOtp().intValue() == otp) {
                return Optional.of(user);
            }
        }
        return Optional.empty();
    }
}
```

EventService.java

Il file "EventService" contiene la definizione di un servizio per la gestione degli eventi all'interno dell'applicazione "Eventify". Questo servizio si occupa di interagire con il database per eseguire operazioni CRUD sugli eventi e implementa diverse funzionalità per la creazione, la modifica e la visualizzazione degli eventi.

Caratteristiche Principali:

1. Iniezione di Dipendenze:
    a. Il servizio fa uso di iniezione delle dipendenze per accedere ad altri componenti dell'applicazione, inclusi il repository degli eventi, il servizio degli utenti, il servizio delle foto e i validatori degli eventi.
2. Creazione di Eventi:
    a. Il metodo "createEvent" è responsabile della creazione di un nuovo evento. Accetta l'ID dell'utente che sta creando l'evento e un oggetto "EventForm" che rappresenta i dettagli dell'evento.
    b. Vengono eseguite diverse validazioni sui dati dell'evento utilizzando un validatore personalizzato.
    c. L'evento viene creato con i dettagli forniti, inclusa la categoria, il creatore, la data e l'ora, la descrizione, il luogo e il titolo.
    d. Le foto caricate per l'evento vengono gestite utilizzando il servizio delle foto.

    e. Una volta creato, l'evento viene salvato nel repository degli eventi e viene restituito un elenco di risposte, che possono includere messaggi di errore o l'ID dell'evento appena creato.
3. Recupero di Eventi:
    a. Il servizio offre metodi per recuperare eventi in base all'ID, per ottenere tutti gli eventi, per cercare eventi per titolo, luogo o categoria, e per ottenere gli eventi creati o registrati da un utente specifico.
4. Aggiornamento ed Eliminazione di Eventi:
    a. Il servizio consente di aggiornare i dettagli di un evento esistente e di eliminarlo tramite metodi dedicati.
5. Eliminazione di Foto da un Evento:
    a. Il servizio fornisce un metodo per eliminare le foto associate a un evento. Vengono eliminate le foto dal repository delle foto utilizzando il servizio delle foto.
6. Validazioni:
    a. Il servizio utilizza i validatori per garantire che i dati dell'evento siano conformi ai requisiti specificati.
7. Gestione degli Oggetti DateTime:
    a. Il servizio gestisce le date e le ore in formato DateTime per rappresentare la data e l'ora degli eventi.
8. Gestione delle Eccezioni:
    a. Il servizio gestisce eccezioni in caso di errori durante la creazione o l'aggiornamento degli eventi.
9. Struttura Modulare:
    a. Il servizio è organizzato in modo modulare e ben strutturato, separando le responsabilità e utilizzando dipendenze iniettate per promuovere la manutenibilità e la scalabilità del codice.
10. Interazione con il Repository:
    a. Il servizio interagisce con il repository degli eventi per eseguire operazioni di persistenza dei dati nel database.

Codice sorgente:

```
package com.eventify.app.service;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.springframework.stereotype.Service;
import org.springframework.web.multipart.MultipartFile;

import com.eventify.app.model.Event;
```

```
import com.eventify.app.model.Photo;
import com.eventify.app.model.User;
import com.eventify.app.model.enums.Categories;
import com.eventify.app.model.json.EventForm;
import com.eventify.app.repository.IEventRepository;
import com.eventify.app.validator.EventValidator;
import com.eventify.app.validator.ObjectsValidator;
import jakarta.transaction.Transactional;
import lombok.RequiredArgsConstructor;

@Service
@RequiredArgsConstructor
public class EventService {

    // Repository per gli eventi
    private final IEventRepository eventRepository;
    // Servizio degli utenti
    private final UserService userService;
    // Servizio per le immagini
    private final PhotoService photoService;
    // Validatore degli eventi
    private final EventValidator eventValidator;
    // Validatore degli oggetti
    private final ObjectsValidator<EventForm> validator;

    // Metodo transazionale per la creazione di un evento
    @Transactional
    public List<Object> createEvent(Long userId, EventForm event) throws Exception {
        // Utilizzo del validatore degli oggetti per verificare il formato dell'evento
        validator.validate(event);

        List<Object> response = new ArrayList<>();

        String errorMessage = null;
        // Verifica se il formato dell'evento è valido, restituendo eventuali errori
        if ((errorMessage = eventValidator.isFormValid(event)) != null) {
            response.add(errorMessage);
            return response;
        }
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss.SSS");

        try {
            LocalDateTime dateTime = LocalDateTime.parse(event.getDateTime(), formatter);
            Event newEvent = Event.builder()
                .category(event.getCategory())
                .creator(userService.getById(userId).get())
                .dateTime(dateTime)
                .description(event.getDescription())
                .place(event.getPlace())
                .title(event.getTitle())
                .isExpired(false)
                .build();
            for (MultipartFile photo : event.getPhotos()) {
                Photo pic = photoService.uploadPhoto(photo);
                pic.setEvent(newEvent);
                photoService.create(pic);

                if (newEvent.getPhotos() == null) {
```

```
                newEvent.setPhotos(new ArrayList<>());
            }
            newEvent.getPhotos().add(pic);
        }
        eventRepository.save(newEvent);
        response.add("Evento creato con successo");
        response.add(newEvent.getId());
        return response;
    } catch (Exception e) {
        System.out.println(e.getMessage());
        response.add(e.getMessage());
        return response;
    }
}

// Restituisce un evento in base all'ID specificato
public Optional<Event> getEventById(Long id) {
    return eventRepository.findById(id);
}

// Restituisce una lista di tutti gli eventi
public List<Event> getAllEvents() {
    return eventRepository.findAll();
}

// Aggiorna un evento
public Event updateEvent(Event event) {
    return eventRepository.save(event);
}

// Elimina un evento in base all'ID specificato
public void deleteEventById(Long id) {
    eventRepository.deleteById(id);
}

// Cerca un evento in base al titolo specificato
public Optional<Event> findByTitle(String title) {
    return eventRepository.findByTitle(title);
}

// Restituisce una lista di eventi in base al luogo specificato
public List<Event> findEventsByPlace(String place) {
    return eventRepository.findByPlace(place);
}

// Restituisce una lista di eventi in base alla categoria specificata
public List<Event> findEventsByCategory(Categories category) {
    return eventRepository.findByCategory(category);
}

// Restituisce una lista di eventi creati da un utente
public List<Event> getEventsCreatedByUser(User user) {
    return eventRepository.findByCreator(user);
}

// Restituisce una lista di eventi a cui un utente ha partecipato
public List<Event> getEventsRegisteredByUser(User user) {
    return eventRepository.findByParticipantsContaining(user);
```

```
    }

    // Elimina le foto associate a un evento
    public boolean deletePhotosByEvent(Event event) {
        List<Photo> photosToBeEliminated = event.getPhotos();

        for (Photo photo : photosToBeEliminated) {
            if (!photoService.delete(photo.getId())) {
                return false;
            }
        }

        return true;
    }
}
```

AuthService.java

Il file "AuthService" rappresenta un servizio all'interno dell'applicazione "Eventify" responsabile dell'autenticazione degli utenti e della gestione delle operazioni di registrazione e accesso all'applicazione. Questo servizio include una serie di funzionalità per la registrazione, il login, la generazione e la gestione dei token di autenticazione, nonché la generazione di codici OTP (One-Time Password) per l'autenticazione a due fattori.

Caratteristiche Principali:

1. Iniezione di Dipendenze:
    a. Il servizio fa uso dell'iniezione delle dipendenze per accedere ad altri componenti dell'applicazione, inclusi il servizio degli utenti, il servizio delle foto, il servizio JWT, il servizio email, il gestore di autenticazione e i validatori.
2. Registrazione di Utenti:
    a. Il metodo "signUp" consente agli utenti di registrarsi all'applicazione. Esso gestisce la validazione dei dati dell'utente e la creazione di un nuovo utente nel sistema.
    b. In caso di errori o duplicazioni (ad esempio, email già registrata), vengono restituiti messaggi di errore appropriati.
3. Autenticazione e Generazione di Token:
    a. Il metodo "signIn" gestisce il processo di accesso degli utenti. Accetta le credenziali di accesso (email e password) e utilizza il gestore di autenticazione per verificare le credenziali.
    b. In caso di successo, vengono generati token di accesso e di refresh, che vengono restituiti come parte di una risposta.

    c. In caso di errore (ad esempio, credenziali errate o utente non registrato), vengono restituiti messaggi di errore appropriati.
4. Generazione di Codici OTP:
    a. Il servizio utilizza il pacchetto "warrenstrange/googleauth" per generare codici OTP (One-Time Password) per l'autenticazione a due fattori.
    b. I codici OTP vengono generati e inviati via email all'utente al momento del login.
5. Gestione del Refresh Token:
    a. Il metodo "refreshToken" permette agli utenti di rigenerare un nuovo token di accesso utilizzando un token di refresh.
    b. Questo meccanismo consente di mantenere l'utente connesso senza dover effettuare nuovamente il login.
6. Gestione delle Eccezioni:
    a. Il servizio gestisce eccezioni in caso di errori durante il processo di autenticazione, registrazione o generazione di token.
7. Validazione dei Dati Utente:
    a. Vengono eseguite validazioni sui dati dell'utente, come l'accettazione dei termini e delle condizioni, la validità dei dati di registrazione e la presenza di foto del profilo.
8. Inclusione dei Dati dell'Utente nell'Autenticazione:
    a. I dati dell'utente, come email ed eventuale data di scadenza del token, vengono inclusi nella risposta dell'autenticazione.
9. Struttura Modulare:
    a. Il servizio è organizzato in modo modulare e ben strutturato, con una chiara separazione delle responsabilità.

Codice sorgente:

```
/**
 * La classe "AuthService" gestisce l'autenticazione e l'autorizzazione degli utenti all'interno dell'applicazione "Eventify".
 * Fornisce funzionalità per la registrazione, il login, la gestione dei token di autenticazione e la generazione di codici OTP per l'autenticazione a due fattori.
 */

package com.eventify.app.service;

import java.util.Date;
import java.util.Optional;

import org.springframework.dao.DataIntegrityViolationException;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
```

```
import org.springframework.stereotype.Service;
import org.springframework.web.util.WebUtils;

import com.eventify.app.model.Photo;
import com.eventify.app.model.User;
import com.eventify.app.model.enums.Role;
import com.eventify.app.model.json.AuthenticationResponse;
import com.eventify.app.model.json.LoginRequest;
import com.eventify.app.model.json.RegisterRequest;
import com.eventify.app.validator.ObjectsValidator;
import com.eventify.app.validator.UserValidator;

import com.warrenstrange.googleauth.GoogleAuthenticator;
import com.warrenstrange.googleauth.GoogleAuthenticatorKey;

import io.jsonwebtoken.io.IOException;
import jakarta.mail.MessagingException;
import jakarta.servlet.http.Cookie;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import jakarta.transaction.Transactional;

import lombok.RequiredArgsConstructor;

@RequiredArgsConstructor
@Service
public class AuthService {

    private final UserService userService;
    private final UserValidator userValidator;
    private final PhotoService photoService;
    private final JwtService jwtService;
    private final EmailService emailService;
    private final AuthenticationManager authManager;
    private final ObjectsValidator<RegisterRequest> validator;

    /**
     * Il metodo "signUp" gestisce la registrazione degli utenti.
     * Verifica la validità dei dati, compresa l'accettazione dei termini e delle condizioni.
     * Se tutto è valido, crea un nuovo utente e restituisce un messaggio di registrazione avvenuta con successo.
     * In caso di errori, restituisce messaggi di errore appropriati.
     *
     * @param request Oggetto contenente i dati della registrazione dell'utente.
     * @return Messaggio di registrazione avvenuta con successo o messaggi di errore.
     * @throws Exception
     */
    @Transactional
    public String signUp(RegisterRequest request) throws Exception {
        validator.validate(request);
        if (!request.isCheckbox()) {
            return "Accettare i termini e le condizioni";
        }
        String errorMessage = null;
        if ((errorMessage = userValidator.isFormValid(request)) != null) {
            return errorMessage;
        }
```

```
        User user = new User(request.getFirstname(), request.getLastname(), request.getDob(), request.getEmail(),
request.getPassword(), null);
        try {
            user.setRole(Role.USER);
            userService.create(user);
        } catch (DataIntegrityViolationException e) {
            return "Email già registrata";
        }
        Photo profilePicture = photoService.uploadPhoto(request.getProfilePicture());
        photoService.create(profilePicture);
        user.setProfilePicture(profilePicture);
        userService.update(user.getId(), user);
        return "Registrazione avvenuta con successo";
    }

    /**
     * Il metodo "signIn" gestisce l'accesso degli utenti all'applicazione.
     * Verifica le credenziali fornite e autentica l'utente.
     * In caso di successo, genera token di accesso e di refresh e restituisce una risposta di autenticazione.
     * In caso di errori, restituisce messaggi di errore appropriati.
     *
     * @param loginRequest Oggetto contenente le credenziali dell'utente (email e password) per il login.
     * @param request Oggetto HttpServletRequest.
     * @param response Oggetto HttpServletResponse.
     * @return Risposta di autenticazione contenente token di accesso, token di refresh ed eventuali messaggi di
errore.
     */
    public AuthenticationResponse signIn(LoginRequest loginRequest, HttpServletRequest request,
HttpServletResponse response) {
        String accessToken = null;
        String refreshToken = null;
        String errorMessage = null;
        Date expirationDate = null;
        try {
            Authentication authentication = authManager.authenticate(new
UsernamePasswordAuthenticationToken(loginRequest.getEmail(), loginRequest.getPassword()));
            SecurityContextHolder.getContext().setAuthentication(authentication);
            Optional<User> user = userService.findByEmail(loginRequest.getEmail());
            if (user.isEmpty()) {
                errorMessage = "Email non registrata";
                return
AuthenticationResponse.builder().error(errorMessage).accessToken(accessToken).refreshToken(refreshToken).
build();
            }

            try {
                String secretKey = generateSecretKey();
                int otp = generateOtp(secretKey);
                emailService.sendSignInEmail(loginRequest.getEmail(), otp);
                user.get().setOtp(otp);
                userService.update(user.get().getId(), user.get());
            } catch (MessagingException e) {
                errorMessage = "Credenziali errate";
                return
AuthenticationResponse.builder().error(errorMessage).accessToken(accessToken).refreshToken(refreshToken).
build();
```

```
            }
            return
AuthenticationResponse.builder().error(errorMessage).accessToken(accessToken).refreshToken(refreshToken).
expirationDate(expirationDate).email(user.get().getEmail()).build();
        } catch (BadCredentialsException e) {
            errorMessage = "Credenziali errate";
            return
AuthenticationResponse.builder().error(errorMessage).accessToken(accessToken).refreshToken(refreshToken).
build();
        }
    }

    /**
     * Il metodo "refreshToken" rigenera un nuovo token di accesso utilizzando un token di refresh.
     * Questa funzionalità permette all'utente di rimanere connesso senza dover effettuare nuovamente il login.
     *
     * @param request Oggetto HttpServletRequest.
     * @param response Oggetto HttpServletResponse.
     * @return Risposta contenente il nuovo token di accesso, token di refresh e data di scadenza.
     * @throws IOException
     */
    public ResponseEntity<AuthenticationResponse> refreshToken(HttpServletRequest request,
HttpServletResponse response) throws IOException {
        String refreshToken;
        String userEmail;
        String accessToken;
        Date expirationDate;
        Cookie tokenCookie = WebUtils.getCookie(request, "refresh_token");

        if (tokenCookie == null) {
            return ResponseEntity.ok(AuthenticationResponse.builder().error("Token di refresh non
valido").refreshToken(null).accessToken(null).build());
        }

        refreshToken = tokenCookie.getValue();
        userEmail = jwtService.extractUsername(refreshToken);

        if (userEmail != null) {
            var user = this.userService.findByEmail(userEmail)
                    .orElseThrow(() -> new UsernameNotFoundException("Utente non trovato con email: " + userEmail));
            if (refreshToken.equals(user.refreshToken)) {
                accessToken = jwtService.generateToken(user);
                Cookie accessTokenCookie = new Cookie("access_token", accessToken);
                accessTokenCookie.setHttpOnly(true);
                accessTokenCookie.setPath("/");
                accessTokenCookie.setSecure(true);
                response.addCookie(accessTokenCookie);
                expirationDate = jwtService.extractExpiration(accessToken);
                return
ResponseEntity.ok(AuthenticationResponse.builder().error(null).refreshToken(null).accessToken(accessToken).e
xpirationDate(expirationDate).build());
            }
        }
        return ResponseEntity.ok(AuthenticationResponse.builder().error("Token di refresh non
valido").refreshToken(null).accessToken(null).build());
    }
```

```
    /**
     * Il metodo "generateOtp" genera un codice OTP (One-Time Password) utilizzando una chiave segreta.
     *
     * @param secretKey Chiave segreta per la generazione del codice OTP.
     * @return Codice OTP generato.
     */
    public Integer generateOtp(String secretKey) {
        GoogleAuthenticator gAuth = new GoogleAuthenticator();
        return gAuth.getTotpPassword(secretKey);
    }

    /**
     * Il metodo "generateSecretKey" genera una chiave segreta per l'autenticazione a due fattori.
     *
     * @return Chiave segreta generata.
     */
    public String generateSecretKey() {
        GoogleAuthenticator gAuth = new GoogleAuthenticator();
        GoogleAuthenticatorKey key = gAuth.createCredentials();
        return key.getKey();
    }
}
```

EmailService.java

La classe "EmailService" fornisce funzionalità per l'invio di email all'interno dell'applicazione "Eventify". È utilizzata per inviare varie notifiche email agli utenti, compresi avvisi di autenticazione, notifiche sugli eventi, notifiche di aggiornamenti del profilo e altro. Ecco una spiegazione dettagliata delle sue funzionalità:

- Il servizio è annotato con `@Service`, indicando che è un componente Spring gestito.
- È iniettato con l'oggetto `JavaMailSender`, che viene utilizzato per inviare email. L'iniezione viene gestita tramite l'annotazione `@Autowired`.

Il servizio contiene diversi metodi per inviare email per scopi diversi:

1. `sendSignInEmail`: Questo metodo invia una email di notifica quando un utente effettua l'accesso. Include una data e un codice OTP (One-Time Password) per l'autenticazione a due fattori. Vengono utilizzate le classi `MimeMessage` e `MimeMessageHelper` per creare e inviare la email.
2. `sendAuthFailure`: Questo metodo invia una email di notifica quando il tentativo di accesso dell'utente fallisce. Include una data e un messaggio di avviso.

3. `sendResetPassword`: Questo metodo invia una email con un codice OTP per il ripristino della password. L'utente può utilizzare questo codice per confermare il ripristino.
4. `sendRefresh2fa`: Questo metodo invia una email con un codice OTP per il ripristino dell'autenticazione a due fattori.
5. `sendCreationEventConfirm`: Questo metodo invia una email di conferma quando un utente ha creato un nuovo evento. Include il titolo dell'evento, la data e l'orario di creazione.
6. `sendRegisterEventConfirm`: Questo metodo invia una email di conferma quando un utente si è registrato con successo per un evento. Include il titolo dell'evento, la data e l'orario di registrazione.
7. `sendUnregisterEventConfirm`: Questo metodo invia una email di conferma quando un utente si è disiscritto con successo da un evento. Include il titolo dell'evento, la data e l'orario di disiscrizione.
8. `sendChangesAdviseAboutEvent`: Questo metodo invia una email di notifica quando ci sono cambiamenti relativi a un evento a cui un utente è registrato. Include il titolo dell'evento e la data/orario in cui è stata rilevata la modifica.
9. `sendChangesAdviseAboutProfile`: Questo metodo invia una email di notifica quando un utente aggiorna con successo il proprio profilo. Include la data e l'orario dell'aggiornamento.
10. `sendEventAbortedAdvice`: Questo metodo invia una email di notifica quando un utente ha cancellato con successo un evento a cui era iscritto. Include il titolo dell'evento, la data e l'orario della cancellazione.
11. `sendEventReminder`: Questo metodo invia una email di promemoria a un utente che è registrato per un evento. Include il titolo dell'evento e un promemoria che l'evento inizierà tra 30 minuti.
12. `sendConfirmChangeEmail`: Questo metodo invia una email per confermare il cambio dell'indirizzo email dell'utente. Include un codice OTP per confermare la modifica.

In ogni metodo, viene creato un oggetto `MimeMessage` e un `MimeMessageHelper` per creare e personalizzare il messaggio email. Sono inclusi dettagli come destinatario, soggetto e testo del messaggio. Successivamente, il messaggio email viene inviato utilizzando l'oggetto `JavaMailSender`.

Codice sorgente:

```
package com.eventify.app.service;

import java.text.SimpleDateFormat;
import java.util.Date;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;
import org.springframework.stereotype.Service;
```

```
import jakarta.mail.MessagingException;
import jakarta.mail.internet.MimeMessage;

@Service
public class EmailService {

    @Autowired
    private final JavaMailSender mailSender;

    // Costruttore: inizializza il servizio con l'oggetto JavaMailSender
    public EmailService(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    // Metodo per inviare una email di notifica dopo un accesso riuscito
    public void sendSignInEmail(String email, Integer otp) throws MessagingException {
        // Creazione di un oggetto MimeMessage per la email
        MimeMessage mimeMessage = mailSender.createMimeMessage();
        // Creazione di un oggetto MimeMessageHelper per la configurazione della email
        MimeMessageHelper messageHelper = new MimeMessageHelper(mimeMessage, true);

        // Formattazione delle date e orari
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
        SimpleDateFormat timeFormat = new SimpleDateFormat("HH:mm:ss");
        String dateStr = dateFormat.format(new Date());
        String timeStr = timeFormat.format(new Date());

        // Impostazione del destinatario, soggetto e testo della email
        messageHelper.setTo(email);
        messageHelper.setSubject("Sign In");
        messageHelper.setText("You have made a new login on " + dateStr + " at " + timeStr + ". If it's not you, you may be at risk of hacking.\nPlease enter the following verification code for 2FA:\n\n" + otp + "\n\n\nHai effettuato un nuovo accesso il " + dateStr + " alle " + timeStr + ", se non sei tu potresti essere sotto esposto a rischi di hackeraggio.\nInserisci il seguente codice per la verifica 2FA\n\n" + otp);

        // Invio della email
        mailSender.send(mimeMessage);
    }

    // Metodo per inviare una email di notifica in caso di autenticazione fallita
    public void sendAuthFailure(String email) throws MessagingException {
        // Simile a sendSignInEmail, ma con messaggio diverso
        // ...
    }

    // Metodo per inviare una email di notifica per il reset della password
    public void sendResetPassword(String email, Integer otp) throws MessagingException {
        // Simile a sendSignInEmail, ma con messaggio diverso
        // ...
    }

    // Metodo per inviare una email di notifica per il refresh dell'autenticazione a due fattori (2FA)
    public void sendRefresh2fa(String email, Integer otp) throws MessagingException {
        // Simile a sendSignInEmail, ma con messaggio diverso
        // ...
```

```
    }

    // Metodo per inviare una email di conferma di creazione di un evento
    public void sendCreationEventConfirm(String email, String title) throws MessagingException {
        // Simile a sendSignInEmail, ma con messaggio diverso
        // ...
    }

    // Metodo per inviare una email di conferma di registrazione a un evento
    public void sendRegisterEventConfirm(String email, String title) throws MessagingException {
        // Simile a sendSignInEmail, ma con messaggio diverso
        // ...
    }

    // Metodo per inviare una email di conferma di deregistrazione da un evento
    public void sendUnregisterEventConfirm(String email, String title) throws MessagingException {
        // Simile a sendSignInEmail, ma con messaggio diverso
        // ...
    }

    // Metodo per inviare una email di notifica di modifiche a un evento
    public void sendChangesAdviseAboutEvent(String email, String title) throws MessagingException {
        // Simile a sendSignInEmail, ma con messaggio diverso
        // ...
    }

    // Metodo per inviare una email di notifica di modifiche al profilo utente
    public void sendChangesAdviseAboutProfile(String email) throws MessagingException {
        // Simile a sendSignInEmail, ma con messaggio diverso
        // ...
    }

    // Metodo per inviare una email di notifica di cancellazione di un evento
    public void sendEventAbortedAdvice(String email, String title) throws MessagingException {
        // Simile a sendSignInEmail, ma con messaggio diverso
        // ...
    }

    // Metodo per inviare una email di promemoria per un evento
    public void sendEventReminder(String email, String title) throws MessagingException {
        // Simile a sendSignInEmail, ma con messaggio diverso
        // ...
    }

    // Metodo per inviare una email di conferma di cambio dell'indirizzo email
    public void sendConfirmChangeEmail(String email, Integer otp) throws MessagingException {
        // Simile a sendSignInEmail, ma con messaggio diverso
        // ...
    }
}
```

FilterService.java

Descrizione

Il file `FilterService` è una classe di servizio all'interno dell'applicazione "Eventify". Questa classe fornisce metodi per filtrare gli eventi in base a diversi criteri come il titolo, il luogo, la categoria, la data di inizio, la data di fine e gli eventi associati a un utente.

Caratteristiche Principali

Dipendenze

- Questa classe è annotata con `@Service`, indicando che è un componente gestito da Spring Framework.
- Usa l'iniezione delle dipendenze per ottenere un'istanza di `EventService` e `UserService` nel costruttore.

Filtraggio degli Eventi

1. `findEventsByTitle(String search, List<Event> events)`: Questo metodo filtra gli eventi in base al titolo. Cerca eventi il cui titolo contiene una sottostringa specificata (case-insensitive) e li restituisce.
2. `findEventsByPlace(String search, List<Event> events)`: Filtra gli eventi in base al luogo. Esegue la ricerca nella città specificata all'interno del luogo dell'evento e restituisce gli eventi corrispondenti.
3. `findEventsByCategory(Categories[] categories, List<Event> events)`: Questo metodo filtra gli eventi in base alla categoria. Confronta la categoria di ciascun evento con un array di categorie specificate e restituisce gli eventi corrispondenti.
4. `findEventsByDateStart(String date, List<Event> events)`: Filtra gli eventi in base alla data di inizio. Gli eventi con data di inizio successiva alla data specificata vengono restituiti.
5. `findEventsByDateEnd(String date, List<Event> events)`: Filtra gli eventi in base alla data di fine. Gli eventi con data di fine precedente alla data specificata vengono restituiti.
6. `findEventsByDateInterval(String start, String end, List<Event> events)`: Questo metodo filtra gli eventi in base a un intervallo di date. Restituisce gli eventi che iniziano dopo la data di inizio specificata e finiscono prima della data di fine specificata.
7. `findEventsByMyEvents(Long userId, List<Event> events)`: Questo metodo restituisce gli eventi creati da un utente specifico. È possibile ottenere gli eventi creati dall'utente utilizzando il servizio `eventService`.

8. `findEventsByRegisteredEvents(Long userId, List<Event> events)`: Questo metodo restituisce gli eventi a cui un utente specifico si è registrato. Utilizza il servizio `eventService` per ottenere gli eventi registrati dall'utente.

Utilizzo

La classe `FilterService` viene utilizzata per filtrare gli eventi in base a criteri specifici. Questi filtri possono essere applicati per aiutare gli utenti a trovare eventi specifici di loro interesse all'interno dell'applicazione "Eventify".

Eccezioni

- Se una data specificata non è nel formato corretto, viene lanciata un'eccezione `IllegalArgumentException` con un messaggio di errore appropriato.

Dipendenze

- `EventService`: Questo servizio è utilizzato per accedere agli eventi all'interno dell'applicazione.
- `UserService`: Questo servizio è utilizzato per accedere agli utenti all'interno dell'applicazione.

Codice sorgente:

```
package com.eventify.app.service;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.springframework.stereotype.Service;
import com.eventify.app.model.Event;
import com.eventify.app.model.User;
import com.eventify.app.model.enums.Categories;

import lombok.AllArgsConstructor;

@Service
@AllArgsConstructor
public class FilterService {

    private final EventService eventService;
    private final UserService userService;

    /**
     * Filtra gli eventi in base al titolo.
     *
     * @param search Parola chiave di ricerca per il titolo.
```

```
 * @param events Elenco di eventi da filtrare.
 * @return Elenco di eventi che corrispondono al criterio di ricerca per il titolo.
 */
public List<Event> findEventsByTitle(String search, List<Event> events) {
    List<Event> filteredEvents = new ArrayList<>();

    for (Event event : events) {
        if (event.getTitle().toLowerCase().contains(search.toLowerCase())) {
            filteredEvents.add(event);
        }
    }
    return filteredEvents;
}

/**
 * Filtra gli eventi in base al luogo.
 *
 * @param search Nome della città di ricerca.
 * @param events Elenco di eventi da filtrare.
 * @return Elenco di eventi che si svolgono nella città specificata.
 */
public List<Event> findEventsByPlace(String search, List<Event> events) {
    List<Event> filteredEvents = new ArrayList<>();
    String city = "";

    for (Event event : events) {
        String[] address = event.getPlace().split(" ");
        city = address[address.length - 2];
        if (city.toLowerCase().equals(search.toLowerCase())) {
            filteredEvents.add(event);
        }
    }
    return filteredEvents;
}

/**
 * Filtra gli eventi in base alla categoria.
 *
 * @param categories Array di categorie per il filtraggio.
 * @param events Elenco di eventi da filtrare.
 * @return Elenco di eventi che corrispondono alle categorie specificate.
 */
public List<Event> findEventsByCategory(Categories[] categories, List<Event> events) {
    List<Event> filteredEvents = new ArrayList<>();

    for (Event event : events) {
        for (Categories category : categories) {
            if (event.getCategory() == category) {
                filteredEvents.add(event);
                break;
            }
        }
    }
    return filteredEvents;
}
```

```
/**
 * Filtra gli eventi in base alla data di inizio.
 *
 * @param date Data di inizio per il filtraggio.
 * @param events Elenco di eventi da filtrare.
 * @return Elenco di eventi che iniziano dopo la data specificata.
 * @throws IllegalArgumentException se il formato della data non è valido.
 */
public List<Event> findEventsByDateStart(String date, List<Event> events) {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss.SSSX");
    date = date.replace("m", "");
    try {
        LocalDateTime dateTime = LocalDateTime.parse(date, formatter);
        List<Event> filteredEvents = new ArrayList<>();
        for (Event event : events) {
            if (event.getDateTime().isAfter(dateTime)) {
                filteredEvents.add(event);
            }
        }
        return filteredEvents;
    } catch (DateTimeParseException e) {
        throw new IllegalArgumentException("Date format is not valid: " + date, e);
    }
}

/**
 * Filtra gli eventi in base alla data di fine.
 *
 * @param date Data di fine per il filtraggio.
 * @param events Elenco di eventi da filtrare.
 * @return Elenco di eventi che terminano prima della data specificata.
 * @throws IllegalArgumentException se il formato della data non è valido.
 */
public List<Event> findEventsByDateEnd(String date, List<Event> events) {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss.SSSX");
    date = date.replace("m", "");
    try {
        LocalDateTime dateTime = LocalDateTime.parse(date, formatter);
        List<Event> filteredEvents = new ArrayList<>();

        for (Event event : events) {
            if (event.getDateTime().isBefore(dateTime)) {
                filteredEvents.add(event);
            }
        }
        return filteredEvents;
    } catch (DateTimeParseException e) {
        throw new IllegalArgumentException("Date format is not valid: " + date, e);
    }
}

/**
 * Filtra gli eventi in base a un intervallo di date.
 *
 * @param start Data di inizio dell'intervallo per il filtraggio.
 * @param end Data di fine dell'intervallo per il filtraggio.
```

```
 * @param events Elenco di eventi da filtrare.
 * @return Elenco di eventi che iniziano dopo la data di inizio e terminano prima della data di fine specificate.
 * @throws IllegalArgumentException se il formato delle date non è valido.
 */
public List<Event> findEventsByDateInterval(String start, String end, List<Event> events) {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss.SSSX");

    start = start.replace("m", "");
    end = end.replace("m", "");

    try {
        LocalDateTime startDateTime = LocalDateTime.parse(start, formatter);
        LocalDateTime endDateTime = LocalDateTime.parse(end, formatter);
        List<Event> filteredEvents = new ArrayList<>();

        for (Event event : events) {
            if (event.getDateTime().isAfter(startDateTime) && event.getDateTime().isBefore(endDateTime)) {
                filteredEvents.add(event);
            }
        }
        return filteredEvents;
    } catch (DateTimeParseException e) {
        throw new IllegalArgumentException("Date format is not valid: " + start + " or " + end, e);
    }
}

/**
 * Restituisce gli eventi creati da un utente specifico.
 *
 * @param userId ID dell'utente per cui si vogliono trovare gli eventi creati.
 * @param events Elenco di eventi da filtrare.
 * @return Elenco di eventi creati dall'utente specifico.
 */
public List<Event> findEventsByMyEvents(Long userId, List<Event> events) {
    Optional<User> userOptional = userService.getById(userId);

    if (userOptional.isPresent()) {
        User user = userOptional.get();
        List<Event> createdEvents = eventService.getEventsCreatedByUser(user);
        return createdEvents;
    }
    return null;
}

/**
 * Restituisce gli eventi a cui un utente specifico è registrato.
 *
 * @param userId ID dell'utente per cui si vogliono trovare gli eventi registrati.
 * @param events Elenco di eventi da filtrare.
 * @return Elenco di eventi a cui l'utente specifico è registrato.
 */
public List<Event> findEventsByRegisteredEvents(Long userId, List<Event> events) {
    Optional<User> userOptional = userService.getById(userId);

    if (userOptional.isPresent()) {
        User user = userOptional.get();
```

```
            List<Event> registeredEvents = eventService.getEventsRegisteredByUser(user);
            return registeredEvents;
        }
        return null;
    }
}
```

EventReminderScheduler.java

Il `EventReminderScheduler` è una classe che gestisce la pianificazione e la rimozione delle notifiche di promemoria per gli eventi. Questa classe sfrutta un thread pool pianificato per inviare promemoria relative agli eventi agli utenti in modo automatico. Di seguito sono spiegate le caratteristiche e il funzionamento di questa classe:

Dipendenze e Variabili di Classe:

- La classe è annotata con `@Component`, il che significa che è gestita dal framework Spring.
- Viene utilizzato il logger SLF4J per registrare eventi e messaggi di log.
- La classe è `@RequiredArgsConstructor`, il che comporta la creazione di un costruttore con parametri per le dipendenze, ovvero `EmailService` e `NotificationService`.
- `ScheduledExecutorService` è utilizzato per pianificare attività asincrone, come l'invio di promemoria.
- `scheduledReminders` è una mappa che tiene traccia delle attività di promemoria pianificate per ciascun evento.

Metodo `scheduleEventReminders`:

- Questo metodo consente di pianificare i promemoria per un evento specifico.
- Calcola la differenza tra l'orario di inizio dell'evento e l'orario attuale.
- Se l'evento inizia tra più di 30 minuti (cioè `delayMillis > 0`), viene creato un task di promemoria.
- Il task di promemoria invia promemoria tramite `emailService` e crea notifiche tramite `notificationService`.

Metodo `unscheduleEventReminders`:

- Questo metodo consente di rimuovere la pianificazione dei promemoria per un evento specifico.
- Verifica se esiste una pianificazione per l'evento nella mappa `scheduledReminders` e la cancella.

Metodo `shutdown`:

- Questo metodo chiude il thread pool pianificato in modo sicuro.
- Attendere al massimo 30 secondi per l'arresto dei thread pianificati.

- In caso di interruzione, verrà eseguito il tentativo di chiusura immediata (`shutdownNow()`).

Metodo `sendEventReminders`:

- Questo metodo effettua l'invio effettivo dei promemoria agli utenti registrati per un evento.
- Viene utilizzato sia `emailService` che `notificationService` per comunicare il promemoria agli utenti.
- Le notifiche vengono inviate sia tramite email che tramite notifiche push.

Utilizzo:

- Questa classe viene utilizzata per pianificare promemoria per gli eventi e garantire che gli utenti vengano avvisati prima dell'inizio dell'evento.
- Il metodo `scheduleEventReminders` viene chiamato quando si crea un nuovo evento o quando si desidera pianificare un promemoria per un evento esistente.
- Il metodo `unscheduleEventReminders` può essere chiamato per annullare la pianificazione del promemoria se l'evento viene cancellato o se il promemoria non è più necessario.
- Il metodo `shutdown` viene chiamato al momento dell'arresto dell'applicazione per garantire la pulizia delle risorse.

Codice sorgente:

```
package com.eventify.app.service;

import java.time.LocalDateTime;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

import org.springframework.stereotype.Component;

import java.time.Duration;
import com.eventify.app.model.Event;
import com.eventify.app.model.User;
import com.fasterxml.jackson.core.JsonProcessingException;

import jakarta.mail.MessagingException;
import lombok.RequiredArgsConstructor;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@Component
@RequiredArgsConstructor
public class EventReminderScheduler {
```

```
    // Logger per la registrazione di eventi e messaggi di log
    private static final Logger logger = LoggerFactory.getLogger(EventReminderScheduler.class);

    // Dipendenze necessarie per il funzionamento del servizio
    private final EmailService emailService;
    private final NotificationService notificationService;

    // Thread pool per la pianificazione delle attività di promemoria
    private ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);

    // Mappa per tenere traccia delle attività di promemoria programmate per ciascun evento
    private final Map<Long, ScheduledFuture<?>> scheduledReminders = new ConcurrentHashMap<>();

    // Metodo per la pianificazione dei promemoria per un evento specifico
    public void scheduleEventReminders(Event event) {
        LocalDateTime eventStartTime = event.getDateTime();
        LocalDateTime now = LocalDateTime.now();

        // Calcola la differenza tra l'orario attuale e l'orario di inizio dell'evento
        Duration timeUntilEvent = Duration.between(now, eventStartTime);

        // Calcola il ritardo in millisecondi (30 minuti prima dell'evento)
        long delayMillis = timeUntilEvent.minusMinutes(30).toMillis();

        if (delayMillis > 0) {
            // Crea un'attività di promemoria
            Runnable reminderTask = () -> {
                try {
                    // Invia il promemoria
                    sendEventReminders(event);
                } catch (JsonProcessingException e) {
                    e.printStackTrace();
                }
            };
            // Pianifica l'attività con il ritardo specificato
            ScheduledFuture<?> scheduledFuture = scheduler.schedule(reminderTask, delayMillis,
TimeUnit.MILLISECONDS);

            // Registra l'attività di promemoria pianificata nella mappa
            scheduledReminders.put(event.getId(), scheduledFuture);

            logger.info("Scheduled reminder for event: {}", event.getId());
        }
    }

    // Metodo per annullare la pianificazione dei promemoria per un evento specifico
    public void unscheduleEventReminders(Event event) {
        Long eventId = event.getId();

        // Recupera l'attività di promemoria pianificata per l'evento
        ScheduledFuture<?> scheduledFuture = scheduledReminders.get(eventId);

        if (scheduledFuture != null) {
            // Cancella l'attività
            scheduledFuture.cancel(false);
```

```
            // Rimuove l'attività di promemoria dalla mappa
            scheduledReminders.remove(eventId);

            logger.info("Unscheduled reminder for event: {}", event.getId());
        }
    }

    // Metodo per arrestare il thread pool in modo sicuro
    public void shutdown() {
        // Arresta il thread pool
        scheduler.shutdown();
        try {
            // Attendere al massimo 30 secondi per l'arresto dei thread pianificati
            if (!scheduler.awaitTermination(30, TimeUnit.SECONDS)) {
                // In caso di interruzione, effettua l'arresto immediato
                scheduler.shutdownNow();
            }
        } catch (InterruptedException e) {
            logger.error("Error while shutting down EventReminderScheduler", e);
        }
    }

    // Metodo per l'invio effettivo dei promemoria
    private void sendEventReminders(Event event) throws JsonProcessingException {
        try {
            // Itera tra i partecipanti all'evento
            for (User participant : event.getParticipants()) {
                // Invia promemoria via email
                emailService.sendEventReminder(participant.getEmail(), event.getTitle());

                // Crea notifiche
                notificationService.createNotification(participant.getId(), event.getId(), "Ti ricordiamo che sta per
cominciare l'evento a cui sei iscritto : ", null);

                // Invia notifiche agli utenti
                notificationService.sendNotificationToUserId(participant.getId());
            }
        } catch (MessagingException e) {
            // Gestisce eventuali errori nell'invio di promemoria
            System.out.println("Error sending reminder for event: " + event.getId());
            logger.error("Error sending reminder for event: {}", event.getId(), e);
        }
    }
}
```

GeocodingService.java

Il codice fornito rappresenta un servizio per la geocodifica, che consente di ottenere le coordinate geografiche (latitudine e longitudine) a partire da un indirizzo utilizzando l'API di Google Maps. Ecco una spiegazione delle caratteristiche principali del codice:

1. Classe di servizio GeocodingService: Questa classe è annotata come un componente di Spring (`@Service`), il che significa che può essere iniettata e utilizzata in altre parti dell'applicazione.
2. Variabile `googleMapsApiKey`: Questa variabile di istanza è annotata con `@Value` e viene iniettata da Spring con il valore dell'API Key di Google Maps da un file di configurazione esterno.
3. Metodo `getCoordinatesFromAddress`: Questo metodo accetta un indirizzo come input e restituisce un array di double contenente la latitudine e la longitudine delle coordinate geografiche corrispondenti all'indirizzo fornito.
4. Utilizzo di Google Maps API: Il metodo crea un oggetto `GeoApiContext` configurato con la chiave API di Google Maps (`googleMapsApiKey`) e utilizza questo contesto per effettuare una richiesta di geocodifica utilizzando `GeocodingApi.geocode()`. Questo metodo restituisce un array di risultati di geocodifica (`GeocodingResult`).
5. Estrazione delle coordinate: Se ci sono risultati di geocodifica validi, il metodo estrae la latitudine e la longitudine dalla prima corrispondenza e le restituisce come un array di double.
6. Gestione delle eccezioni: Il metodo gestisce eventuali eccezioni, ad esempio errori di connessione a Internet o errori di geocodifica, stampando l'errore nella console e restituendo `null`.

In generale, questa classe fornisce un servizio utile per ottenere le coordinate geografiche a partire da un indirizzo, il che può essere utile in un'applicazione che richiede la visualizzazione o la gestione di informazioni basate sulla posizione geografica. La chiave API di Google Maps è necessaria per utilizzare questo servizio e deve essere configurata nel file di configurazione dell'applicazione.

Codice sorgente:

```
package com.eventify.app.service;

import com.google.maps.GeoApiContext;
import com.google.maps.GeocodingApi;
import com.google.maps.model.GeocodingResult;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

@Service
public class GeocodingService {

    // Inietta la chiave API di Google Maps dalla configurazione esterna
    @Value("${google.maps.apikey}")
    private String googleMapsApiKey;

    // Metodo per ottenere le coordinate geografiche da un indirizzo
    public double[] getCoordinatesFromAddress(String address) {
        try {
```

```
            // Crea un contesto GeoApiContext con la chiave API di Google Maps
            GeoApiContext context = new GeoApiContext.Builder()
                    .apiKey(googleMapsApiKey)
                    .build();

            // Effettua una richiesta di geocodifica utilizzando l'indirizzo fornito
            GeocodingResult[] results = GeocodingApi.geocode(context, address).await();

            // Verifica se ci sono risultati di geocodifica validi
            if (results != null && results.length > 0) {
                // Estrae la latitudine e la longitudine dalla prima corrispondenza
                double latitude = results[0].geometry.location.lat;
                double longitude = results[0].geometry.location.lng;
                // Restituisce le coordinate come un array di double
                return new double[]{latitude, longitude};
            }
        } catch (Exception e) {
            // Gestisce eventuali eccezioni, ad esempio errori di geocodifica o di connessione
            e.printStackTrace();
        }

        // Restituisce null in caso di errori o mancanza di risultati
        return null;
    }
}
```

JwtService.java

Caratteristiche Principali

1. Generazione di Token JWT e Refresh Token: Il servizio `JwtService` consente di generare due tipi di token: il token JWT standard e il refresh token. Questi token vengono utilizzati per autenticare gli utenti e consentire loro l'accesso alle risorse dell'applicazione.
2. Validità Temporale dei Token: Sia il token JWT che il refresh token hanno un periodo di validità temporale specifico. Il token JWT standard ha un periodo di validità di 24 ore (86400000 millisecondi), mentre il refresh token ha una validità più lunga di 7 giorni (604800000 millisecondi).
3. Estrazione delle Informazioni dal Token: Il servizio consente di estrarre informazioni utili dal token, ad esempio il nome utente, la data di scadenza e altre informazioni personalizzate.
4. Validazione dei Token: È possibile verificare se un token è valido confrontando il nome utente estratto dal token con l'utente autenticato e verificando se il token è scaduto.
5. Generazione dei Token: I token vengono generati utilizzando il framework JSON Web Token (JWT) con una chiave segreta (SECRET_KEY) e l'algoritmo di firma HMAC-SHA256.

Metodi Principali

1. `generateToken(UserDetails userDetails)`: Questo metodo genera un token JWT standard per un utente specifico (`UserDetails`). Il token JWT include il nome utente, le informazioni aggiuntive (extraClaims), la data di creazione (IssuedAt) e la data di scadenza (Expiration).
2. `generateToken(Map<String, Object> extraClaims, UserDetails userDetails)`: Questo metodo consente di generare un token JWT personalizzato con informazioni aggiuntive specifiche, oltre ai dettagli dell'utente. Questo è utile per includere dati personalizzati nel token.
3. `generateRefreshToken(UserDetails userDetails)`: Questo metodo genera un refresh token, che ha una validità temporale più lunga rispetto al token JWT standard. Il refresh token è utilizzato per ottenere un nuovo token JWT dopo la scadenza del token standard.
4. `isTokenValid(String jwt, UserDetails user)`: Questo metodo verifica se un token JWT è valido per un utente specifico. Controlla se il nome utente nel token corrisponde all'utente autenticato e se il token è scaduto.
5. `extractUsername(String jwt)`: Questo metodo estrae il nome utente dal token JWT.
6. `extractExpiration(String jwt)`: Questo metodo estrae la data di scadenza dal token JWT.

Chiave Segreta (SECRET_KEY)

Il servizio utilizza una chiave segreta (`SECRET_KEY`) per firmare e verificare i token. La chiave segreta deve essere mantenuta segreta e configurata nel servizio. È importante che questa chiave non venga esposta o condivisa pubblicamente, in quanto è utilizzata per garantire l'integrità e la sicurezza dei token.

Sicurezza

È importante sottolineare che la sicurezza delle applicazioni basate su token JWT dipende dalla corretta gestione delle chiavi segrete e dalla gestione dei token. Le chiavi segrete dovrebbero essere conservate in modo sicuro e non dovrebbero essere esposte in modo non sicuro nel codice sorgente. Inoltre, i token dovrebbero essere protetti e crittografati quando necessario per garantire la sicurezza delle informazioni sensibili.

Il servizio `JwtService` è un componente chiave nell'implementazione dell'autenticazione e dell'autorizzazione in un'applicazione Spring, offrendo la generazione, la gestione e la validazione dei token JWT.
Codice sorgente:

```
package com.eventify.app.service;

import java.security.Key;
```

```
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;

@Service
public class JwtService {

    // Chiave segreta per la firma e la verifica dei token
    private static final String SECRET_KEY =
"3cf7ff7e1f14e47d43bab57653a80512459106b707946a88098975a07ac381fd";

    // Durata di validità del token JWT (24 ore in millisecondi)
    private static final long jwtExpiration = 86400000;

    // Durata di validità del refresh token (7 giorni in millisecondi)
    private static final long refreshExpiration = 604800000;

    // Estrae il nome utente dal token JWT
    public String extractUsername(String jwt) {
        return extractClaim(jwt, Claims::getSubject);
    }

    // Verifica se un token è valido per un utente specifico
    public boolean isTokenValid(String jwt, UserDetails user) {
        final String username = extractUsername(jwt);
        return username.equals(user.getUsername()) && !isTokenExpired(jwt);
    }

    // Verifica se un token è scaduto
    private boolean isTokenExpired(String jwt) {
        return extractExpiration(jwt).before(new Date());
    }

    // Estrae la data di scadenza dal token JWT
    public Date extractExpiration(String jwt) {
        return extractClaim(jwt, Claims::getExpiration);
    }

    // Estrae le informazioni dal token JWT
    public <T> T extractClaim(String jwt, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(jwt);
        return claimsResolver.apply(claims);
    }

    // Estrae tutte le informazioni dal token JWT
    public Claims extractAllClaims(String jwt) {
```

```
        return Jwts.parserBuilder().setSigningKey(getSignInKey()).build().parseClaimsJws(jwt).getBody();
    }

    // Ottiene la chiave segreta come oggetto Key
    private Key getSignInKey() {
        byte[] keyBytes = Decoders.BASE64.decode(SECRET_KEY);
        return Keys.hmacShaKeyFor(keyBytes);
    }

    // Genera un token JWT standard per un utente
    public String generateToken(UserDetails userDetails) {
        return generateToken(new HashMap<>(), userDetails);
    }

    // Genera un token JWT personalizzato con informazioni aggiuntive
    public String generateToken(Map<String, Object> extraClaims, UserDetails userDetails) {
        return buildToken(extraClaims, userDetails, jwtExpiration);
    }

    // Genera un refresh token per un utente
    public String generateRefreshToken(UserDetails userDetails) {
        return buildToken(new HashMap<>(), userDetails, refreshExpiration);
    }

    // Costruisce un token JWT con le informazioni specificate
    private String buildToken(Map<String, Object> extraClaims, UserDetails userDetails, long expiration) {
        return Jwts.builder()
            .setClaims(extraClaims)
            .setSubject(userDetails.getUsername())
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis() + expiration))
            .signWith(getSignInKey(), SignatureAlgorithm.HS256)
            .compact();
    }
}
```

LogoutService.java

Questa classe è responsabile per la gestione delle operazioni di logout degli utenti all'interno dell'applicazione. Di seguito, una descrizione delle principali caratteristiche del codice:

1. LogoutService come LogoutHandler: La classe `LogoutService` implementa l'interfaccia `LogoutHandler`, il che significa che è incaricata di gestire le operazioni di logout quando richieste dal framework di sicurezza di Spring durante l'eliminazione delle credenziali di autenticazione.
2. Costruttore con Dipendenze: Il costruttore di `LogoutService` accetta due dipendenze principali: `JwtService` e `UserService`. Queste dipendenze vengono iniettate nel servizio attraverso l'annotazione `@RequiredArgsConstructor`, semplificando così l'iniezione delle dipendenze.

3. Logout e Rimozione dei Cookie: Nel metodo `logout`, vengono eseguite le seguenti operazioni:
    a. Sono create due oggetti `Cookie` per i token di accesso (`access_token`) e di refresh (`refresh_token`). Questi cookie vengono inizializzati con il valore `null`, impostati come `HttpOnly` per motivi di sicurezza, impostati sul percorso radice (`/`) e impostati a una durata massima di `0`, il che li rende obsoleti e quindi rimossi dal browser dell'utente.
    b. Viene estratto il valore del refresh token dalla cookie `refresh_token`, che sarà utilizzato per identificare l'utente interessato dal logout.
4. Rimozione del Refresh Token dall'Utente: Una volta ottenuto il valore del refresh token e quindi l'indirizzo email associato all'utente, viene effettuata una query al servizio `UserService` per trovare l'utente corrispondente a quell'indirizzo email. Il refresh token dell'utente viene quindi impostato su `null`, il che revoca la validità del refresh token, impedendo l'uso futuro dello stesso.
5. Pulizia del Contesto di Sicurezza: Infine, il contesto di sicurezza viene pulito utilizzando `SecurityContextHolder.clearContext()`. Questa operazione revoca l'identità dell'utente autenticato, garantendo che non possa più effettuare azioni non autorizzate.

Codice sorgente:

```
package com.eventify.app.service;

import jakarta.servlet.http.Cookie;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;

import java.util.Optional;

import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.authentication.logout.LogoutHandler;
import org.springframework.stereotype.Service;

import com.eventify.app.model.User;

@Service
@RequiredArgsConstructor
public class LogoutService implements LogoutHandler {

    private final JwtService jwtService; // Servizio per gestire i token JWT
    private final UserService userService;  // Servizio per gestire gli utenti

    @Override
    public void logout(HttpServletRequest request, HttpServletResponse response, Authentication authentication) {
        // Crea un nuovo cookie "access_token" e lo setta a null, rendendolo obsoleto
        Cookie accessTokenCookie = new Cookie("access_token", null);
        accessTokenCookie.setHttpOnly(true); // Imposta il cookie come HttpOnly per sicurezza
```

```
        accessTokenCookie.setPath("/"); // Imposta il percorso radice per il cookie
        accessTokenCookie.setMaxAge(0); // Imposta la durata a 0 per rimuovere il cookie
        response.addCookie(accessTokenCookie); // Aggiunge il cookie alla risposta HTTP

        // Crea un nuovo cookie "refresh_token" e lo setta a null, rendendolo obsoleto
        Cookie refreshTokenCookie = new Cookie("refresh_token", null);
        refreshTokenCookie.setHttpOnly(true); // Imposta il cookie come HttpOnly per sicurezza
        refreshTokenCookie.setPath("/"); // Imposta il percorso radice per il cookie
        refreshTokenCookie.setMaxAge(0); // Imposta la durata a 0 per rimuovere il cookie
        response.addCookie(refreshTokenCookie); // Aggiunge il cookie alla risposta HTTP

        // Estrae il valore del refresh token dalla cookie "refresh_token"
        String refreshToken = refreshTokenCookie.getValue();
        String userEmail = jwtService.extractUsername(refreshToken); // Estrae l'indirizzo email dall'ID utente

        // Cerca l'utente associato all'indirizzo email
        Optional<User> user = this.userService.findByEmail(userEmail);

        // Imposta il refresh token dell'utente su null, revocando la validità del refresh token
        user.get().setRefreshToken(null);

        // Aggiorna le informazioni dell'utente nel database
        userService.update(user.get().getId(), user.get());

        // Pulisce il contesto di sicurezza, revocando l'identità dell'utente autenticato
        SecurityContextHolder.clearContext();
    }
}
```

NotificationService.java

Il `NotificationService` è un componente del sistema che gestisce le notifiche degli utenti. Ecco una descrizione delle sue principali caratteristiche:

1. Gestione delle Notifiche: Il servizio offre funzionalità per la gestione delle notifiche, inclusa la creazione, l'aggiornamento, l'eliminazione e il recupero delle notifiche. Le notifiche sono oggetti che contengono un messaggio e possono essere associate a eventi specifici o ad altre azioni degli utenti.
2. Interazione con il Repository: Il servizio interagisce con un repository (`INotificationRepository`) per memorizzare e recuperare le notifiche. Questo repository è responsabile della persistenza dei dati delle notifiche.
3. Creazione di Notifiche: Il metodo `createNotification` consente di creare notifiche. Questo metodo richiede l'ID dell'utente a cui inviare la notifica, l'ID dell'evento (opzionale), un messaggio e un titolo (opzionale). A seconda del tipo di azione, il messaggio viene personalizzato per riflettere l'azione dell'utente.

4. Visualizzazione delle Notifiche dell'Utente: Il metodo `showUserNotification` consente di recuperare tutte le notifiche associate a un utente specifico. Queste notifiche vengono ordinate per data e ora decrescenti in modo che le notifiche più recenti siano visualizzate per prime.
5. Invio di Notifiche agli Utenti: Il metodo `sendNotificationToUserId` viene utilizzato per inviare notifiche specifiche a un utente. Questo è spesso utilizzato per inviare notifiche tramite Server-Sent Events (SSE) ai client interessati.
6. Recupero di Notifiche per un Utente: Il metodo `getNotificationsByUserId` consente di recuperare tutte le notifiche associate a un utente specifico. Questo metodo può essere utilizzato per recuperare tutte le notifiche di un utente per ulteriori elaborazioni o visualizzazioni.
7. Personalizzazione delle Notifiche: Le notifiche vengono personalizzate in base all'azione dell'utente. Ad esempio, se un utente si iscrive o annulla l'iscrizione a un evento, il messaggio della notifica riflette questa azione e menziona l'evento corrispondente.
8. Gestione della Lettura delle Notifiche: Le notifiche possono contenere un attributo `isRead` per tenere traccia se sono state lette o meno. La gestione di questa funzionalità non è evidente nel codice, ma potrebbe essere implementata nell'applicazione.

Codice sorgente:

```
package com.eventify.app.service;

import com.eventify.app.model.Event;
import com.eventify.app.model.Notification;
import com.eventify.app.model.User;
import com.eventify.app.model.json.NotificationForm;
import com.eventify.app.repository.INotificationRepository;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

import lombok.RequiredArgsConstructor;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;

import org.springframework.stereotype.Service;

@Service
@RequiredArgsConstructor
public class NotificationService {

    private final INotificationRepository notificationRepository;  // Repository per gestire le notifiche
    private final UserService userService;  // Servizio per gestire gli utenti
```

```
    private final EventService eventService;  // Servizio per gestire gli eventi

    // Restituisce una notifica in base all'ID
    public Optional<Notification> getNotificationById(Long id) {
        return notificationRepository.findById(id);
    }

    // Restituisce tutte le notifiche
    public List<Notification> getAllNotifications() {
        return notificationRepository.findAll();
    }

    // Aggiorna una notifica
    public Notification updateNotification(Notification event) {
        return notificationRepository.save(event);
    }

    // Cancella una notifica in base all'ID
    public void deleteNotificationById(Long id) {
        notificationRepository.deleteById(id);
    }

    // Crea una nuova notifica e la salva nel repository
    public void createNotification(Long userId, Long eventId, String message, String title) {
        User user = userService.getById(userId).get();
        Event event = null;

        if (eventId != null) {
            event = eventService.getEventById(eventId).get();
        }

        if (message == "Subscribed for creator") {
            message = "The user " + user.getFirstname() + " " + user.getLastname() + " is subscribed to your event: "
+ event.getTitle();

            if (eventId != null) {
                user = userService.getById(event.getCreator().getId()).get();
            }
        } else if (message == "Unsubscribed for creator") {
            message = "The user " + user.getFirstname() + " " + user.getLastname() + " is unsubscribed to your
event: " + event.getTitle();

            if (eventId != null) {
                user = userService.getById(event.getCreator().getId()).get();
            }
        } else {
            if (eventId != null) {
                message += event.getTitle();
            } else if (title != null) {
                message += title;
            }
        }

        Notification notification = new Notification(user, message);
        notificationRepository.save(notification);
    }

    // Mostra tutte le notifiche di un utente
```

```
    public List<Notification> showUserNotification(Long userId) {
        List<Notification> notifications = getAllNotifications();
        List<Notification> notificheUser = new ArrayList<>();

        for (Notification notification : notifications) {
            if (notification.getUser().getId() == userId) {
                notificheUser.add(notification);
            }
        }

        notificheUser.sort(Comparator.comparing(Notification::getDateTime).reversed());
        return notificheUser;
    }

    // Invia notifiche ad un utente specifico
    public boolean sendNotificationToUserId(Long userId) throws JsonProcessingException {
        List<Notification> notificationList = getAllNotifications();
        List<NotificationForm> resultNotifications = new ArrayList();

        for (Notification notification : notificationList) {
            if (notification.getUser().getId() == userId) {
                resultNotifications.add(NotificationForm.builder()
                .isRead(notification.getIsRead())
                .message(notification.getMessage())
                .dateTime(notification.getDateTime())
                .build());
            }
        }

        resultNotifications.sort(Comparator.comparing(NotificationForm::getDateTime).reversed());
        ObjectMapper objectMapper = new ObjectMapper();
        String json = objectMapper.writeValueAsString(resultNotifications);

        // Invia l'evento Server-Sent Events (SSE) con le notifiche JSON
        SseService.sendSseEventToClients(String.valueOf(userId), json);

        return true;
    }

    // Ottiene tutte le notifiche di un utente in base al suo ID
    public List<Notification> getNotificationsByUserId(Long userId) {
        List<Notification> allNotifications = notificationRepository.findAll();
        List<Notification> notificationResponse = new ArrayList<>();

        for (Notification notification: allNotifications) {
            if (notification.getUser().getId() == userId) {
                notificationResponse.add(notification);
            }
        }

        return notificationResponse;
    }
}
```

PhotoService.java

Il servizio `PhotoService` è responsabile della gestione delle immagini e delle operazioni associate alle immagini. Ecco una descrizione delle principali caratteristiche e funzionalità del servizio:

1. Gestione di Foto: Il servizio gestisce oggetti di tipo `Photo`, che rappresentano le immagini caricate nell'applicazione.
2. Recupero di Immagini: Il metodo `getAll` consente di recuperare tutte le immagini presenti nell'applicazione. Restituisce un elenco di oggetti `Photo`.
3. Recupero di Immagine per ID: Il metodo `getById` permette di recuperare una specifica immagine in base al suo ID. Restituisce un oggetto `Optional<Photo>`, che può essere vuoto se l'immagine non è presente.
4. Creazione di Immagini: Il metodo `create` consente di creare nuove immagini. Prende un oggetto `Photo` come input, lo salva nel repository e restituisce l'immagine creata.
5. Aggiornamento di Immagini: Il metodo `update` permette di aggiornare un'immagine esistente. Prende l'ID dell'immagine da aggiornare e un nuovo oggetto `Photo` con le informazioni aggiornate. Restituisce un oggetto `Optional<Photo>` che rappresenta l'immagine aggiornata. Se l'immagine non esiste, restituisce un `Optional` vuoto.
6. Eliminazione di Immagini: Il metodo `delete` consente di eliminare un'immagine in base al suo ID. Restituisce un booleano che indica se l'eliminazione è avvenuta con successo.
7. Caricamento di Immagini: Il metodo `uploadPhoto` viene utilizzato per caricare nuove immagini nell'applicazione. Prende come input un oggetto `MultipartFile`, comunemente utilizzato per l'upload di file tramite richieste HTTP. Questo metodo esegue la pulizia del nome del file e verifica che non contenga sequenze di percorso non valide. Quindi, crea un nuovo oggetto `Photo` con le informazioni del file e i dati binari dell'immagine.

In generale, il servizio `PhotoService` fornisce funzionalità per la gestione delle immagini, inclusa la creazione, l'aggiornamento, l'eliminazione e il recupero delle immagini. Inoltre, offre un metodo per caricare nuove immagini, garantendo che i nomi dei file siano sicuri. Questo servizio è utile in applicazioni in cui è richiesta la gestione delle immagini, come gallerie fotografiche o upload di foto per gli utenti.

Codice sorgente:

```
package com.eventify.app.service;

import java.util.Optional;
import org.springframework.stereotype.Service;
```

```
import org.springframework.util.StringUtils;
import org.springframework.web.multipart.MultipartFile;

import com.eventify.app.service.interfaces.IPhotoService;
import com.eventify.app.model.Photo;
import com.eventify.app.repository.IPhotoRepository;

@Service("mainService")
public class PhotoService implements IPhotoService {

    private IPhotoRepository iPhotoRepository;

    public PhotoService(IPhotoRepository iPhotoRepository) {
        this.iPhotoRepository = iPhotoRepository;
    }

    // Recupera tutte le immagini
    @Override
    public Iterable<Photo> getAll() {
        return iPhotoRepository.findAll();
    }

    // Recupera un'immagine per ID
    @Override
    public Optional<Photo> getById(Long id) {
        return iPhotoRepository.findById(id);
    }

    // Crea una nuova immagine
    @Override
    public Photo create(Photo photo) {
        return iPhotoRepository.save(photo);
    }

    // Aggiorna un'immagine esistente
    @Override
    public Optional<Photo> update(Long id, Photo photo) {
        Optional<Photo> foundPhoto = iPhotoRepository.findById(id);

        if (foundPhoto.isEmpty()) {
            return Optional.empty();
        }

        // Aggiorna le informazioni dell'immagine
        foundPhoto.get().setPhotoName(photo.getPhotoName());
        foundPhoto.get().setPhotoType(photo.getPhotoType());
        foundPhoto.get().setData(photo.getData());
        foundPhoto.get().setIsDeleted(photo.getIsDeleted());

        // Salva le modifiche nel repository
        iPhotoRepository.save(foundPhoto.get());

        return foundPhoto;
    }

    // Elimina un'immagine per ID
    @Override
    public Boolean delete(Long id) {
```

```
        Optional<Photo> foundPhoto = iPhotoRepository.findById(id);
        if (foundPhoto.isEmpty()) {
            return false;
        }

        // Elimina l'immagine dal repository
        iPhotoRepository.delete(foundPhoto.get());
        return true;
    }

    // Carica una nuova immagine da un oggetto MultipartFile
    public Photo uploadPhoto(MultipartFile image) throws Exception {
        String fileName = StringUtils.cleanPath(image.getOriginalFilename());

        try {
            if (fileName.contains("..")) {
                throw new Exception("Il nome del file contiene sequenze di percorso non valide " + fileName);
            }

            // Crea un nuovo oggetto Photo con le informazioni del file
            Photo photo = new Photo(image.getContentType(), fileName, Boolean.FALSE, image.getBytes());
            return photo;
        } catch (Exception e) {
            throw new Exception("Impossibile salvare il file: " + fileName);
        }
    }
}
```

SseService.java

- Questa classe gestisce la comunicazione SSE tra il server e i client. SSE è un protocollo unidirezionale che consente al server di inviare dati in modo asincrono ai client su una singola connessione persistente.
- La classe è responsabile della gestione degli "emitters" (emettitori), che rappresentano i client connessi.
- I metodi principali della classe includono `addEmitter`, `removeEmitter`, e `sendSseEventToClients`, che vengono utilizzati per gestire la registrazione degli emettitori, la rimozione degli emettitori e l'invio di eventi SSE ai client.

Caratteristiche:

1. addEmitter: Questo metodo consente di registrare un emettitore associandolo a un ID univoco del sottoscrittore (client). Gli emettitori vengono memorizzati in una mappa (`emitters`) in cui la chiave è l'ID del sottoscrittore e il valore è l'emettitore stesso.
2. removeEmitter: Questo metodo rimuove un emettitore associato a un determinato ID del sottoscrittore. Questo avviene eliminando l'associazione dall'elenco degli emettitori.

3. sendSseEventToClients: Questo metodo invia un evento SSE a un client specifico. Prima di farlo, verifica che ci siano client connessi e che l'emettitore associato all'ID del sottoscrittore specifico esista. Se nessun client è connesso o se l'emettitore non esiste, vengono registrati messaggi di avviso appropriati. Se l'emettitore esiste, il metodo invia l'evento SSE ai client utilizzando il tipo di media `MediaType.APPLICATION_JSON`.
4. Logger: La classe utilizza un logger per registrare messaggi di avviso e errori relativi all'invio di eventi SSE o all'assenza di client connessi.
5. Mappa degli emettitori: La classe utilizza una mappa (`emitters`) per tracciare gli emettitori associati agli ID dei sottoscrittori. Questa mappa consente una gestione efficace degli emettitori e delle connessioni dei client.

Codice sorgente:

```
package com.eventify.app.service;

import org.springframework.http.MediaType;
import org.springframework.web.servlet.mvc.method.annotation.SseEmitter;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Logger;

public class SseService {
    // Logger per la registrazione di messaggi
    private static final Logger logger = Logger.getLogger(SseService.class.getName());

    // Mappa per la gestione degli emettitori associati agli ID dei sottoscrittori (client)
    private static final Map<String, SseEmitter> emitters = new HashMap<>();

    // Aggiunge un emettitore associato a un ID di sottoscrittore
    public static void addEmitter(String subscriberId, SseEmitter emitter) {
        emitters.put(subscriberId, emitter);
    }

    // Rimuove un emettitore associato a un ID di sottoscrittore
    public static void removeEmitter(String subscriberId) {
        emitters.remove(subscriberId);
    }

    // Invia un evento SSE a un cliente specifico
    public static void sendSseEventToClients(String subscriberId, String data) {
        // Verifica se ci sono client connessi
        if (emitters.size() == 0) {
            logger.warning("Nessun client connesso per inviare l'evento!");
        }
        // Ottiene l'emettitore associato all'ID del sottoscrittore
        var emitter = emitters.get(subscriberId);
        // Se l'emettitore non esiste, registra un messaggio di avviso
        if (emitter == null) {
            logger.warning("Nessun client con l'ID del sottoscrittore " + subscriberId + " trovato!");
            return;
```

```
        }
        try {
            // Invia l'evento SSE ai client con il tipo di media "APPLICATION_JSON"
            emitter.send(data, MediaType.APPLICATION_JSON);
        } catch (IOException e) {
            // Registra un messaggio di avviso in caso di errore nell'invio dell'evento
            logger.warning("Errore nell'invio dell'evento al client: " + e.getMessage());
        }
    }
}
```

Validatori

EventValidator.java

La classe `EventValidator` fornisce una serie di metodi per la validazione degli eventi e dei file immagine associati ad essi in un'applicazione. Ecco una descrizione tecnica delle caratteristiche del codice:

Validazione del Modulo Evento:

- Il metodo `isFormValid` è utilizzato per verificare la validità di un oggetto `EventForm`, che rappresenta i dati relativi a un evento.
- Controlla se i campi obbligatori, come il titolo, la descrizione, il luogo, la categoria, le immagini e la data/ora dell'evento, sono stati forniti. Se uno qualsiasi di questi campi è vuoto o mancante, viene restituito un messaggio di errore.

Validazione della Data e Ora:

- La data e l'ora dell'evento vengono validate per garantire che siano correttamente formattate come stringhe e che la data e l'ora siano almeno 24 ore nel futuro rispetto all'istante attuale. Altrimenti, viene restituito un messaggio di errore.

Validazione del Nome:

- Il metodo `isValidName` verifica se una stringa passata corrisponde al modello di espressione regolare definito in `NAME_REGEX`. Questo modello accetta solo lettere minuscole e maiuscole (senza spazi o caratteri speciali).

Validazione del File Immagine:

- Il metodo `isImageFile` verifica se un file è un'immagine controllando i primi byte del file. Supporta il rilevamento di immagini nei formati JPEG (byte header: 0xFFD8FFE0) e PNG (byte header: 0x89504E47).
- Se il file non è valido o il rilevamento dà esito negativo, viene restituito `false`.

Utility per il Controllo dell'Intestazione del File:

- Il metodo privato `startsWith` verifica se una serie di byte inizia con un particolare prefisso, utilizzato per la validazione dei file immagine.

Questa classe è utile per garantire che gli eventi inseriti nell'applicazione rispettino determinati criteri di validità, inclusi campi obbligatori, formato data/ora, nome e tipi di file delle immagini consentiti. Le funzionalità di validazione aiutano a mantenere la coerenza dei dati e la sicurezza dell'applicazione.

Codice sorgente:

```
package com.eventify.app.validator;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;
import org.springframework.stereotype.Component;
import org.springframework.web.multipart.MultipartFile;

import com.amazonaws.util.IOUtils;
import com.eventify.app.model.json.EventForm;

@Component
public class EventValidator {

   // Espressione regolare per il nome consentito
   private static final String NAME_REGEX = "^[A-Za-z]+$";

   // Intestazioni binarie dei formati di immagine supportati (JPEG e PNG)
   private static final byte[][] SUPPORTED_IMAGE_MAGIC_NUMBERS = {
      {(byte) 0xFF, (byte) 0xD8, (byte) 0xFF, (byte) 0xE0}, // JPEG
      {(byte) 0x89, (byte) 0x50, (byte) 0x4E, (byte) 0x47}  // PNG
   };

   // Verifica la validità del modulo EventForm
   public String isFormValid(EventForm event) {
      if (event.getTitle() == null || event.getTitle().isEmpty() ||
         event.getDescription() == null || event.getDescription().isEmpty() ||
         event.getPlace() == null || event.getPlace().isEmpty() ||
         event.getTitle().isEmpty() || event.getCategory() == null ||
         event.getPhotos().isEmpty() || event.getDateTime() == null)
      {
         return "All fields must be filled";
      }

      try {
         // Ottiene l'istante temporale attuale
         LocalDateTime currentTime = LocalDateTime.now();
         // Definisce un formato per la data e l'ora
         DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss.SSS");
         // Converte la data e l'ora dell'evento in oggetto LocalDateTime
         LocalDateTime dateTime = LocalDateTime.parse(event.getDateTime(), formatter);

         // Verifica se la data e l'ora dell'evento sono almeno 24 ore nel futuro
         if (dateTime.isBefore(currentTime.plus(24, ChronoUnit.HOURS))) {
```

```
                return "Event date and time must be at least 24 hours from now";
            }
        } catch (Exception e) {
            System.out.println(e.getMessage());
            return "Invalid date";
        }

        // Verifica se ciascun file nell'elenco delle immagini è un file immagine valido
        for (MultipartFile photo : event.getPhotos()) {
            if (!isImageFile(photo)) {
                return "Invalid image file format in event photos. Only image files are allowed.";
            }
        }
        return null;
    }

    // Verifica se una stringa rappresenta un nome valido
    public boolean isValidName(String name) {
        return name.matches(NAME_REGEX);
    }

    // Verifica se un MultipartFile è un file immagine valido
    public boolean isImageFile(MultipartFile file) {
        if (file == null || file.isEmpty()) {
            return false;
        }
        try {
            // Legge i primi byte del file
            byte[] fileBytes = IOUtils.toByteArray(file.getInputStream());
            // Verifica se i byte corrispondono a uno dei formati di immagine supportati
            for (byte[] magicNumber : SUPPORTED_IMAGE_MAGIC_NUMBERS) {
                if (startsWith(fileBytes, magicNumber)) {
                    return true;
                }
            }
            return false;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    // Verifica se un array di byte inizia con un dato prefisso
    private boolean startsWith(byte[] array, byte[] prefix) {
        if (array.length < prefix.length) {
            return false;
        }
        for (int i = 0; i < prefix.length; i++) {
            if (array[i] != prefix[i]) {
                return false;
            }
        }
        return true;
    }
}
```

ObjectsValidator.java

1. Classe Generica: Il servizio è implementato come una classe generica `ObjectsValidator<T>`, dove `T` è il tipo dell'oggetto da validare. Questo consente di riutilizzare il servizio per diversi tipi di oggetti.
2. Gestione delle Validazioni: Il servizio utilizza l'API di validazione di Jakarta Bean Validation (JavaX Validation) per eseguire le validazioni sull'oggetto `objectToValidate`.
3. Inizializzazione del Validator: Viene inizializzato un oggetto `ValidatorFactory` utilizzando `Validation.buildDefaultValidatorFactory()`, e da questo viene ottenuto un oggetto `Validator`. Questo oggetto `Validator` è responsabile dell'esecuzione delle validazioni.
4. Metodo `validate`: Il metodo `validate(T objectToValidate)` accetta un oggetto da validare. Questo metodo esegue le validazioni specifiche per l'oggetto `objectToValidate` utilizzando il `Validator`.
5. Rilevamento delle Violazioni: Le violazioni, ovvero i messaggi di errore o le regole di validazione non rispettate, vengono raccolte in un insieme (`Set`) di oggetti `ConstraintViolation`. Le violazioni vengono rilevate utilizzando il metodo `validator.validate(objectToValidate)`.
6. Gestione degli Errori: Se vengono rilevate delle violazioni (cioè l'insieme di violazioni non è vuoto), il servizio raccoglie i messaggi di errore dalla lista di violazioni. Questi messaggi di errore rappresentano le regole di validazione non rispettate.
7. Sollevamento di un'Eccezione: Viene sollevata un'eccezione `ObjectValidationException`. Questa eccezione contiene i messaggi di errore e il nome semplice della classe dell'oggetto validato. Sollevando questa eccezione, il servizio segnala un fallimento nella validazione.
8. Logging degli Errori: I messaggi di errore vengono anche stampati sulla console utilizzando `System.out.println(error)`. Questa operazione di logging può essere personalizzata in base alle esigenze specifiche dell'applicazione.

In sintesi, il servizio `ObjectsValidator` offre una solida struttura per eseguire la validazione degli oggetti in base alle regole di validazione specificate e segnalare eventuali violazioni mediante un'eccezione personalizzata. Questo aiuta a garantire che gli oggetti siano conformi alle regole di validazione prima di procedere con ulteriori operazioni.

Codice sorgente:

```
package com.eventify.app.validator;

import com.eventify.app.exception.ObjectValidationException;
import jakarta.validation.ConstraintViolation;
```

```
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.ValidatorFactory;
import java.util.Set;
import java.util.stream.Collectors;
import org.springframework.stereotype.Service;

@Service
public class ObjectsValidator<T> {

  // Factory e Validator per le operazioni di validazione
  private final ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
  private final Validator validator = factory.getValidator();

  // Metodo per validare un oggetto generico
  public void validate(T objectToValidate) {
    // Esegue le validazioni e raccoglie eventuali violazioni
    Set<ConstraintViolation<T>> violations =  validator.validate(objectToValidate);

    if (!violations.isEmpty()) {
      // Raccoglie i messaggi di errore dalle violazioni
      Set<String> errorMsg = violations
          .stream()
          .map(ConstraintViolation::getMessage)
          .collect(Collectors.toSet());
       for (String error : errorMsg) {
         // Stampa i messaggi di errore sulla console
         System.out.println(error);
       }
      // Solleva un'eccezione di validazione personalizzata
      throw new ObjectValidationException(errorMsg, objectToValidate.getClass().getSimpleName());
    }
  }
}
```

UserValidator.java

La classe `UserValidator` è responsabile di validare le informazioni degli utenti, sia durante la registrazione che durante la modifica del profilo. Il validatore applica diverse regole per garantire che i dati dell'utente siano coerenti e sicuri.

Regole di Validazione

1. Nome Utente: I campi `firstname` e `lastname` devono contenere solo caratteri alfabetici e non possono essere vuoti.
2. Data di Nascita: La data di nascita (`dob`) deve essere fornita e l'utente deve avere almeno 18 anni.
3. Indirizzo Email: L'indirizzo email deve essere valido e seguire il formato `user@example.com`.
4. Immagine Profilo: L'immagine del profilo deve essere un file di immagine valido. Sono supportati i formati JPEG e PNG.

5. Conferma Password: Durante la registrazione e la modifica del profilo, è necessario confermare la password. La password deve contenere almeno 8 caratteri e rispettare i criteri di forza:
    - Deve contenere almeno un carattere maiuscolo.
    - Deve contenere almeno un carattere speciale tra `@`, `#`, `$`, `%`, `^`, `&`, `+`, `=`, `!`.
    - Deve contenere almeno un numero.

Metodi di Validazione

La classe `UserValidator` offre due metodi di validazione:

1. `isFormValid(RegisterRequest registerRequest)`: Questo metodo valida un oggetto di tipo `RegisterRequest`, che viene utilizzato durante la registrazione. Controlla tutte le regole di validazione sopra descritte e restituisce un messaggio di errore se una qualsiasi di queste regole non è soddisfatta.
2. `isFormValid(User user, MultipartFile imageFile, String confirmPassword)`: Questo metodo valida un utente esistente, il file dell'immagine del profilo e la conferma della password. È utilizzato per la modifica del profilo e segue le stesse regole di validazione descritte sopra.
3. Altri metodi di supporto sono disponibili per validare il nome, la forza della password, l'indirizzo email e il formato del file immagine.

Regole sui File di Immagine

Il validatore verifica il formato dei file di immagine confrontando i primi byte del file con firme magiche associate ai formati JPEG e PNG. Questo garantisce che solo file di immagine validi siano accettati come immagini del profilo.

Codice sorgente:

```
package com.eventify.app.validator;

import java.time.LocalDate;
import java.time.Period;
import java.time.ZoneId;
import java.util.Date;

import org.springframework.stereotype.Component;
import org.springframework.web.multipart.MultipartFile;

import com.amazonaws.util.IOUtils;
import com.eventify.app.model.User;
import com.eventify.app.model.json.RegisterRequest;

@Component
public class UserValidator {

    // Espressioni regolari per le validazioni
```

```
    private static final String NAME_REGEX = "^[A-Za-z]+$";
    private static final String PASSWORD_REGEX = "^(?=.*[0-9])(?=.*[A-Z])(?=.*[@#$%^&+=!]).*$";
    private static final String EMAIL_REGEX = "^[A-Za-z0-9+_.-]+@([A-Za-z0-9.-]+)\\.[A-Za-z]{2,4}$";

    // Firme magiche per i formati di immagine supportati
    private static final byte[][] SUPPORTED_IMAGE_MAGIC_NUMBERS = {
        {(byte) 0xFF, (byte) 0xD8, (byte) 0xFF, (byte) 0xE0},  // Formato JPEG
        {(byte) 0x89, (byte) 0x50, (byte) 0x4E, (byte) 0x47}   // Formato PNG
    };

    // Metodo per validare i dati di registrazione
    public String isFormValid(RegisterRequest registerRequest) {
        if (registerRequest.getFirstname() == null || registerRequest.getFirstname().isEmpty() ||
            registerRequest.getLastname() == null || registerRequest.getLastname().isEmpty() ||
            registerRequest.getEmail() == null || registerRequest.getEmail().isEmpty() ||
            registerRequest.getPassword() == null || registerRequest.getPassword().isEmpty() ||
            registerRequest.getConfirmPassword() == null || registerRequest.getConfirmPassword().isEmpty() ||
            registerRequest.getDob() == null || registerRequest.getProfilePicture() == null ||
registerRequest.isCheckbox() == false
            ) {
            return "All fields must be filled";
        }

        // Calcola l'età dalladata di nascita
        Date dobDate = registerRequest.getDob();
        LocalDate dob = dobDate.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();
        LocalDate today = LocalDate.now();
        Period age = Period.between(dob, today);

        // Validazione del nome
        if (!isValidName(registerRequest.getFirstname()) || !isValidName(registerRequest.getLastname())) {
            return "Firstname and lastname must contain only alphabetic characters";
        }

        // Verifica del formato dell'immagine
        if (!isImageFile(registerRequest.getProfilePicture())) {
            return "Invalid image file format. Only image files are allowed.";
        }

        // Validazione dell'indirizzo email
        if (!isValidEmail(registerRequest.getEmail())) {
            return "Invalid email address";
        }

        // Verifica dell'età minima
        if (age.getYears() < 18) {
            return "You must be at least 18 years old";
        }

        // Verifica delle password corrispondenti
        if (!registerRequest.getPassword().equals(registerRequest.getConfirmPassword())) {
            return "Passwords do not match";
        }

        // Verifica della lunghezza minima della password
        if (registerRequest.getPassword().length() < 8)  {
```

```
        return "Password is less than 8 characters";
    }

    // Verifica della forza della password
    if (!isStrongPassword(registerRequest.getPassword())) {
        return "Password must contain at least: an uppercase char, a special character, and a number";
    }

    return null;  // Nessun errore
}

// Metodo per validare un utente e i dati di modifica del profilo
public String isFormValid(User user, MultipartFile imageFile, String confirmPassword) {
    if (user.getFirstname() == null || user.getFirstname().isEmpty() ||
        user.getLastname() == null || user.getLastname().isEmpty() ||
        user.getEmail() == null || user.getEmail().isEmpty() ||
        user.getPassword() == null || user.getPassword().isEmpty() ||
        user.getDob() == null || imageFile == null || confirmPassword == null) {
        return "All fields must be filled";
    }

    // Calcola l'età dalla data di nascita
    Date dobDate = user.getDob();
    LocalDate dob = dobDate.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();
    LocalDate today = LocalDate.now();
    Period age = Period.between(dob, today);

    // Validazione del nome
    if (!isValidName(user.getFirstname()) || !isValidName(user.getLastname())) {
        return "Firstname and lastname must contain only alphabetic characters";
    }

    // Verifica del formato dell'immagine
    if (!isImageFile(imageFile)) {
        return "Invalid image file format. Only image files are allowed.";
    }

    // Validazione dell'indirizzo email
    if (!isValidEmail(user.getEmail())) {
        return "Invalid email address";
    }

    // Verifica dell'età minima
    if (age.getYears() < 18) {
        return "You must be at least 18 years old";
    }

    // Verifica delle password corrispondenti
    if (!user.getPassword().equals(confirmPassword)) {
        return "Passwords do not match";
    }

    // Verifica della lunghezza minima della password
    if (user.getPassword().length() < 8)  {
        return "Password is less than 8 characters";
    }
```

```
        // Verifica della forza della password
        if (!isStrongPassword(user.getPassword())) {
            return "Password must contain at least: an uppercase char, a special character, and a number";
        }

        return null;  // Nessun errore
    }

    // Metodo per verificare se una stringa corrisponde all'espressione regolare per il nome
    public boolean isValidName(String name) {
        return name.matches(NAME_REGEX);
    }

    // Metodo per verificare se una stringa corrisponde all'espressione regolare per una password forte
    public boolean isStrongPassword(String password) {
        return password.matches(PASSWORD_REGEX);
    }

    // Metodo per verificare se una stringa corrisponde all'espressione regolare per un indirizzo email valido
    public boolean isValidEmail(String email) {
        return email.matches(EMAIL_REGEX);
    }

    // Metodo per verificare se un file è un'immagine valida
    public boolean isImageFile(MultipartFile file) {
        if (file == null || file.isEmpty()) {
            return false;
        }
        try {
            byte[] fileBytes = IOUtils.toByteArray(file.getInputStream());
            for (byte[] magicNumber : SUPPORTED_IMAGE_MAGIC_NUMBERS) {
                if (startsWith(fileBytes, magicNumber)) {
                    return true;
                }
            }
            return false;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    // Metodo per verificare se un array di byte inizia con una data sequenza di byte
    private boolean startsWith(byte[] array, byte[] prefix) {
        if (array.length < prefix.length) {
            return false;
        }
        for (int i = 0; i < prefix.length; i++) {
            if (array[i] != prefix[i]) {
                return false;
            }
        }
        return true;
    }
}
```

AppApplication.java

Questo è il codice di avvio dell'applicazione principale, che fa parte di un'applicazione Spring Boot. Ecco una spiegazione dettagliata:

`@SpringBootApplication`: Questa annotazione è un'annotazione di Spring Boot che combina diverse annotazioni come `@Configuration`, `@EnableAutoConfiguration`, `@ComponentScan`, ecc., per avviare l'applicazione Spring. Indica che questa classe è la classe di avvio dell'applicazione e contiene anche il metodo `main`.
Il metodo `main(String[] args)`: Questo è il punto di ingresso principale dell'applicazione. Viene eseguito quando si avvia l'applicazione. In questo metodo, vengono eseguite le seguenti operazioni:

- `SpringApplication.run(AppApplication.class, args)`: Questa riga avvia l'applicazione Spring utilizzando la classe `AppApplication` come configurazione principale e passa gli argomenti della riga di comando `args`. Questo metodo restituisce un oggetto `ApplicationContext`, che è il contesto dell'applicazione Spring.
- `EventReminderScheduler eventReminderScheduler = context.getBean(EventReminderScheduler.class)`: Utilizza l'oggetto `ApplicationContext` per ottenere un bean chiamato `EventReminderScheduler`. Questo indica che l'applicazione fa uso di un componente `EventReminderScheduler` per la pianificazione degli eventi.
- `Runtime.getRuntime().addShutdownHook(...)`: Questo codice aggiunge un hook di spegnimento (shutdown hook) che verrà eseguito quando l'applicazione viene arrestata. Nel nostro caso, viene eseguito un thread che chiama il metodo `shutdown()` del `EventReminderScheduler`. Questo assicura che il servizio di pianificazione degli eventi venga arrestato correttamente prima di chiudere l'applicazione.

Codice sorgente:

```
package com.eventify.app;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

import com.eventify.app.service.EventReminderScheduler;

@SpringBootApplication
public class AppApplication {

    public static void main(String[] args) {
```

```
        // Creazione di un contesto dell'applicazione Spring
        ApplicationContext context = SpringApplication.run(AppApplication.class, args);

        // Ottenimento di un bean EventReminderScheduler dal contesto
        EventReminderScheduler eventReminderScheduler = context.getBean(EventReminderScheduler.class);

        // Aggiunta di un hook di spegnimento per gestire la chiusura dell'applicazione
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            eventReminderScheduler.shutdown();
        }));
    }

}
```

Application.properties

```
# Attiva o disattiva la modalità debug dell'applicazione
debug=true

# Imposta il livello di registrazione dei log per il modulo di sicurezza Spring su "DEBUG"
logging.level.org.springframework.security=DEBUG

# Configurazione per Thymeleaf, un motore di template
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
spring.thymeleaf.cache=false

# Configurazione per il supporto del caricamento di file nei servlet Spring
spring.servlet.multipart.enabled=true
spring.servlet.multipart.file-size-threshold=2KB
spring.servlet.multipart.max-file-size=200MB
spring.servlet.multipart.max-request-size=215MB

# Configurazione dell'URL del database
spring.datasource.url=jdbc:postgresql://localhost:5432/eventify
spring.datasource.username=

# Configurazione per l'accesso al database, tra cui la password
spring.datasource.password=

# Proprietà Hibernate per lo schema di default
spring.jpa.properties.hibernate.default_schema=eventify

# Dialect per il database PostgreSQL
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect

# Mostra le query SQL nell'output dei log
spring.jpa.show-sql=true

# Inizializza il database ogni volta che l'applicazione viene eseguita
spring.sql.init.mode=always

# Imposta il livello di registrazione dei log per il modulo Spring Web su "DEBUG"
logging.level.org.springframework.web=DEBUG
```

```
# Imposta il livello di registrazione dei log per Hibernate sugli errori
logging.level.org.hibernate=ERROR

# Genera automaticamente il DDL (Data Definition Language)
spring.jpa.generate-ddl=true

# Configurazione per l'invio di email tramite SMTP
spring.mail.host=smtp.gmail.com
spring.mail.port=465
spring.mail.username=eventify.service@gmail.com
spring.mail.password=hxus xzqj dwqu lgja

# Protocollo SMTP sicuro
spring.mail.protocol=smtps

# Abilita l'autenticazione SMTP
spring.mail.properties.mail.smtp.auth=true

# Abilita il TLS per la connessione SMTP
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.smtp.starttls.required=true

# Abilita SSL per la connessione SMTP
spring.mail.properties.mail.smtp.ssl.enable=true

# Porta su cui verrà eseguito il server
server.port=8443

# Configurazione SSL per la connessione HTTPS
server.ssl.key-store=classpath:eventify.p12
server.ssl.key-store-password=cUb0bGCJXmy1UvSNCu2trKKnsFbseUv8BpS033X4bsN022gaaU
server.ssl.key-store-type=pkcs12
server.ssl.key-alias=eventify

# Imposta il livello di registrazione dei log per il modulo Spring Web Servlet su "DEBUG"
logging.level.org.springframework.web.servlet.DispatcherServlet=DEBUG

# Chiave API per le API di Google Maps
google.maps.apikey = AIzaSyDdOFro2FmBdFO-dZXpJTYMUgsT9WJ1xfQ

# Abilita la protezione DDoS
security.ddos.enabled=true

# Imposta la soglia per il rilevamento degli attacchi HTTP Flood
security.ddos.http.threshold=1000

# Imposta la soglia per il rilevamento degli attacchi TCP Flood
security.ddos.tcp.threshold=10000

# Imposta la soglia per il rilevamento degli attacchi UDP Flood
security.ddos.udp.threshold=100000

# Imposta il tempo di attesa per il rilevamento degli attacchi DDoS
security.ddos.timeout=10s

# Imposta la dimensione del buffer per il rilevamento degli attacchi DDoS
security.ddos.buffer.size=1000
```

```
# Imposta la politica per il blocco degli attacchi DDoS
security.ddos.block.policy=reject

# Abilita tutti gli endpoint di gestione
management.endpoints.web.exposure.include=*
```

Pom.xml

Il file XML fornito è un file di configurazione del sistema di gestione delle dipendenze Maven (POM - Project Object Model) per un progetto Spring Boot. Ecco una descrizione tecnica delle diverse sezioni e delle principali caratteristiche del file:

1. **Dichiarazione XML:**
   ```
   <?xml version="1.0" encoding="UTF-8"?>
   ```

   Questa dichiarazione specifica che il file è un documento XML con codifica UTF-8.

2. **Elemento `<project>`:**
   Questo è l'elemento radice del file POM e contiene tutte le informazioni relative al progetto e alle sue dipendenze.

3. **`<modelVersion>`:**
   ```
   <modelVersion>4.0.0</modelVersion>
   ```

   Questo elemento specifica la versione del modello POM utilizzato. In questo caso, viene utilizzata la versione 4.0.0.

4. **`<parent>`:**
   ```
   <parent> <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>3.1.5</version> <relativePath/> <!-- lookup parent from
   repository --> </parent>
   ```

   Questa sezione fa riferimento al POM genitore da cui ereditare le configurazioni di base. In questo caso, il progetto utilizza il genitore `spring-boot-starter-parent` con la versione 3.1.5.

5. **`<groupId>`, `<artifactId>`, `<version>`:**
   ```
   <groupId>com.eventify</groupId> <artifactId>app</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   ```

Questi elementi specificano l'identificatore univoco del progetto, il nome dell'artefatto (JAR o WAR), e la versione del progetto.

6. **`<name>` e `<description>`:**
   `<name>app</name> <description>Demo project for Spring Boot</description>`

   Questi elementi forniscono il nome e la descrizione del progetto.

7. **`<properties>`:**
   `<properties> <java.version>17</java.version> <frontend-maven-plugin.version>1.12.1</frontend-maven-plugin.version> </properties>`

   Questa sezione definisce alcune proprietà personalizzate utilizzate nel progetto, come la versione di Java e la versione di un plugin Maven.

8. **`<dependencies>`:**
   Questa sezione elenca le dipendenze (librerie) necessarie per il progetto. Alcuni esempi includono librerie come Spring Boot, Commons FileUpload, Google Auth, Amazon AWS SDK, Lombok, PostgreSQL, Spring Security, Thymeleaf, e molte altre.

9. **`<build>`:**
   Questa sezione contiene le configurazioni relative al processo di compilazione del progetto. In particolare, contiene il plugin di Spring Boot Maven, che semplifica la creazione di un eseguibile avviabile del progetto Spring Boot.

Questo file POM è una parte essenziale del progetto Maven e definisce le dipendenze, la struttura del progetto e molte altre configurazioni necessarie per la compilazione e l'esecuzione dell'applicazione Spring Boot. Ogni dipendenza elencata sotto `<dependencies>` rappresenta una libreria utilizzata nel progetto e verrà scaricata automaticamente da Maven quando il progetto viene compilato e le dipendenze sono risolte.

Codice sorgente:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>3.1.5</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>
	<groupId>com.eventify</groupId>
	<artifactId>app</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>app</name>
```

```
<description>Demo project for Spring Boot</description>
<properties>
	<java.version>17</java.version>
	<frontend-maven-plugin.version>1.12.1</frontend-maven-plugin.version>
</properties>
<dependencies>
	<dependency>
		<groupId>commons-fileupload</groupId>
		<artifactId>commons-fileupload</artifactId>
		<version>1.5</version>
	</dependency>
	<dependency>
		<groupId>com.warrenstrange</groupId>
		<artifactId>googleauth</artifactId>
		<version>1.5.0</version>
	</dependency>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-web</artifactId>
	</dependency>
	<dependency>
	  <groupId>com.amazonaws</groupId>
	  <artifactId>aws-java-sdk-s3</artifactId>
	  <version>1.12.576</version>
	</dependency>
	<dependency>
		<groupId>io.jsonwebtoken</groupId>
		<artifactId>jjwt-api</artifactId>
		<version>0.11.5</version>
	</dependency>
	<dependency>
		<groupId>io.jsonwebtoken</groupId>
		<artifactId>jjwt-impl</artifactId>
		<version>0.11.5</version>
	</dependency>
	<dependency>
		<groupId>io.jsonwebtoken</groupId>
		<artifactId>jjwt-jackson</artifactId>
		<version>0.11.5</version>
	</dependency>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-test</artifactId>
		<scope>test</scope>
	</dependency>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-thymeleaf</artifactId>
	</dependency>
	<dependency>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-data-jpa</artifactId>
	</dependency>
	<dependency>
		<groupId>org.postgresql</groupId>
		<artifactId>postgresql</artifactId>
		<scope>runtime</scope>
	</dependency>
```

```
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-security</artifactId>
		</dependency>
		<dependency>
			<groupId>org.projectlombok</groupId>
			<artifactId>lombok</artifactId>
			<scope>annotationProcessor</scope>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-mail</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-validation</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-websocket</artifactId>
		</dependency>
		<dependency>
				<groupId>com.google.maps</groupId>
				<artifactId>google-maps-services</artifactId>
				<version>2.2.0</version> <!-- Sostituisci con l'ultima versione disponibile -->
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-actuator</artifactId>
			<version>3.1.5</version>
		</dependency>
	</dependencies>
	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>
</project>
```

FRONTEND

Questa documentazione fornisce una panoramica completa dell'organizzazione e dell'architettura del nostro sistema. Nel corso di questa documentazione, esploreremo i framework, i componenti chiave e il servizio `redirect.service` utilizzati per realizzare l'applicazione.

Framework Utilizzati

Il nostro progetto fa uso di una serie di framework che costituiscono la base su cui l'applicazione è stata costruita. Di seguito sono elencati i principali framework che hanno contribuito al successo del nostro progetto:

-Angular: Angular è il framework principale su cui si basa l'applicazione. Questo potente framework consente lo sviluppo di un'applicazione web robusta, fornendo una struttura solida per la gestione della logica.

-Ionic: Ionic è un framework che ci ha permesso di realizzare una versione ibrida dell'applicazione, garantendo una coerente esperienza utente su diverse piattaforme.

-Material Design (Angular Material): Abbiamo integrato Angular Material per realizzare un'interfaccia utente moderna e coerente. Grazie ai componenti grafici reattivi e di alta qualità, l'esperienza utente è stata notevolmente migliorata.

-Swiper.js: Swiper.js è stato utilizzato per implementare caroselli e gallerie scorrevoli all'interno dell'applicazione, migliorando ulteriormente l'interazione e la visualizzazione di contenuti dinamici.

Componenti Chiave

I componenti dell'applicazione sono i mattoni fondamentali che costituiscono il nostro sistema. Ciascun componente svolge un ruolo specifico e contribuisce in modo significativo all'esperienza globale dell'utente. Di seguito, sono elencati i componenti chiave che giocano un ruolo fondamentale all'interno del nostro progetto:

-event-blank: Questo componente rappresenta una card vuota per l'aggiunta di nuovi eventi all'interno dell'applicazione.

-event-card: Il componente `event-card` è utilizzato per visualizzare un singolo evento nella event-board, consentendo agli utenti di accedere alle informazioni e interagire con l'evento.

-event-info: `event-info` fornisce informazioni dettagliate sugli eventi, aiutando gli utenti a ottenere una comprensione completa dei dettagli e permettendogli di iscriversi.

-event-board: La vista principale dell'applicazione che mostra un elenco di eventi disponibili. Utilizziamo il componente `event-board` come punto centrale per l'accesso agli eventi.

-event-creation: Questo componente consente agli utenti di creare nuovi eventi in modo intuitivo, fornendo un'interfaccia per l'inserimento dei dati.

-filter-form: Il componente `filter-form` offre un modulo per la creazione di filtri, facilitando la ricerca di eventi specifici.

-footer: Il componente `footer` fornisce informazioni aggiuntive e collegamenti utili per gli utenti.

-forgot-password: Permette agli utenti di avviare il processo di reimpostazione della password in caso di smarrimento delle credenziali d'accesso.

-register: `register` consente agli utenti di registrarsi per un account all'interno dell'applicazione.

-login: Il componente `login` offre un'interfaccia per l'accesso agli utenti registrati.

-header: `header` è il componente dell'intestazione dell'applicazione che fornisce accesso rapido alle funzionalità chiave.

-home: La vista iniziale dell'applicazione, che offre una panoramica dello scopo della applicazione.

-modify-events: Questo componente permette agli utenti autorizzati di apportare modifiche agli eventi esistenti.

-profile: Il componente `profile` mostra il profilo dell'utente, consentendo di visualizzare e modificare le informazioni personali.

-reset-password: `reset-password` gestisce il processo di reimpostazione della password dopo la richiesta da parte dell'utente.

-two-fa-login: Questo componente fornisce un'opzione di accesso a due fattori per una maggiore sicurezza.

Servizio di Redirect (`redirect.service`)

Il nostro progetto utilizza il servizio `redirect.service` per gestire in modo efficiente i reindirizzamenti e controllare lo stato di autenticazione dell'utente. Questo servizio svolge un ruolo cruciale all'interno dell'applicazione.

Funzionalità del Servizio

-Reindirizzamenti Dinamici: Il servizio `redirect.service` consente di effettuare reindirizzamenti dinamici in base alle azioni dell'utente. Ad esempio, è in grado di reindirizzare l'utente alla vista `event-board` applicando filtri diversi in base al tasto premuto.

-Controllo dell'Autenticazione: Il servizio è in grado di verificare se l'utente è autenticato o meno. Questa funzionalità è fondamentale per regolare l'accesso alle diverse parti dell'applicazione.

Analisi dei componenti

Di seguito, l'analisi dei singoli componenti. Svolta prima sul file html e poi sul TypeScript.

event-blank:

```
<mat-card class="blank-card" (click)="createEvent()" >
<mat-card-content class="center-content">
<mat-icon class="giant-plus">add</mat-icon>
</mat-card-content>
</mat-card>
```

<mat-card>: Questo elemento rappresenta una carta grafica Material Design, che funge da contenitore visivo per l'area dedicata all'aggiunta di eventi. La classe CSS blank-card può essere utilizzata per personalizzare l'aspetto della carta.

<mat-card-content>: Questo elemento rappresenta il contenuto principale della carta e ospita il simbolo "plus" per aggiungere eventi.

<mat-icon>: Questo elemento rappresenta un'icona Material Design di un "plus", che suggerisce all'utente la possibilità di aggiungere un nuovo evento. Il clic su questa icona attiva la funzione createEvent().

```
import { Component } from '@angular/core';
import { Router } from '@angular/router';

@Component({
selector: 'app-event-blank',
templateUrl: './event-blank.component.html',
styleUrls: ['./event-blank.component.css']
})
export class EventBlankComponent {

constructor(private router: Router) {}

createEvent() {
this.router.navigate(['/event-creation']);
}
}
```

Il componente è decorato con il decorator @Component che definisce il selettore (selector), il modello (templateUrl) e il foglio di stile (styleUrls) associati al componente.
Il costruttore del componente accetta un'istanza del servizio Router, che verrà utilizzato per gestire il reindirizzamento dell'utente quando si desidera creare un nuovo evento.
Il metodo **createEvent()** viene attivato quando l'utente fa clic sull'area del EventBlankComponent. Questo metodo utilizza il servizio Router per reindirizzare l'utente alla pagina di creazione di eventi, consentendo all'utente di iniziare il processo di creazione di un nuovo evento.

event-card:

```
<a [routerLink]="['/event-info', event.id]">
<mat-card class="event-card" [style.background-image]="'url(' +
event.imageURL + ')'">
<mat-card-header>
<mat-card-title class="title">{{ event.title }}</mat-card-title>
```

```
<p class="category" [ngStyle]="{'background-color':
categoryColors[event.category]}">{{ event.category }}</p>
</mat-card-header>
<mat-card-content>
<p class="description">{{ event.description }}</p>
</mat-card-content>
<mat-card-footer class="footer">
<p class="address">{{ event.address }}</p>
<p class="time">{{ event.date | date: 'dd/MM/yyyy HH:mm' }}</p>
</mat-card-footer>
</mat-card>
</a>
```

<a [routerLink]...>: Questo elemento `<a>` è utilizzato come collegamento per reindirizzare l'utente alla vista `event-info` relativa all'evento. Il reindirizzamento avviene quando l'utente fa clic sull'elemento.

<mat-card>: Questo elemento rappresenta una carta grafica Material Design che contiene i dettagli dell'evento. Il colore di sfondo della carta può essere personalizzato in base all'URL dell'immagine dell'evento.

<mat-card-header>: Questa sezione contiene il titolo dell'evento e la categoria dell'evento, con il colore di sfondo associato alla categoria.

<mat-card-content>: Questa sezione contiene la descrizione dell'evento.

<mat-card-footer>: Questa sezione contiene l'indirizzo dell'evento e la data, formattata come `dd/MM/yyyy HH:mm`.

```
import { Component, ElementRef, AfterViewInit, Renderer2, Input, OnInit
} from '@angular/core';
import { AxiosService } from '../axios.service';
import { AxiosResponse } from 'axios';

type CategoryColors = {
[key: string]: string;
};

@Component({
selector: 'app-event-card',
templateUrl: './event-card.component.html',
styleUrls: ['./event-card.component.css']
```

```
})
export class EventCardComponent implements AfterViewInit, OnInit {
@Input() event: any;

categoryColors: CategoryColors = {
PARTY: '#ff0000', // Rosso
GYM: '#ff8000' ,// Arancio
ROLEGAME: '#653239', // Giallo
SPORT: '#27AE60', // Verde
MEETING: '#083D77', // Blu
CONFERENCE: '#FFBA08', // Ciano
NETWORKING: '#7f5539', // Giallo
HOBBY: '#f032e6', // Magenta
MUSIC: '#fabed4', // Pink
BUSINESS: '#34495E', // Grigio scuro
FOOD: '#911eb4', // Viola
NIGHTLIFE: 'black', // Arancione
HEALTH: '#27AE60', // Verde
HOLIDAYS: '#3498DB' // Blu
};

constructor(private elementRef: ElementRef, private renderer:
Renderer2, private axiosService: AxiosService) {}

ngOnInit() {
if (!this.event) {
// Se l'evento non è stato fornito, crea un evento di esempio
this.event = {
id: '',
title: '',
category: '',
description: '',
address: '',
date: new Date().toISOString(),
// image:
};
}
}

ngAfterViewInit() {
// Ottieni l'elemento con width: auto; (l'elemento HTML del componente)
const element = this.elementRef.nativeElement;
```

```
// Calcola la larghezza effettiva in pixel
const width = element.offsetWidth;

// Puoi fare qualcosa con la larghezza, ad esempio, stamparla nella
console
console.log(`Larghezza effettiva: ${width}px`);
}
}
```

Il componente è decorato con il decorator @Component che definisce il selettore (selector), il modello (templateUrl) e il foglio di stile (styleUrls) associati al componente.
Il componente accetta un input event che rappresenta i dettagli dell'evento da visualizzare.
Il categoryColors oggetto definisce i colori associati alle diverse categorie di eventi. Questi colori vengono utilizzati per personalizzare lo sfondo del badge di categoria.
Il componente implementa l'interfaccia AfterViewInit e OnInit per eseguire azioni specifiche al caricamento del componente.

event-info:

```
<section>
<mat-card class="event-card">
<mat-card-header>
<mat-card-title class="title">{{ event.title }}</mat-card-title>
</mat-card-header>
<swiper-container style="--swiper-navigation-color: #fff;
--swiper-pagination-color: #fff" class="mySwiper"
navigation="true" autoplay-delay="2500"
autoplay-disable-on-interaction="false" slides-per-view='auto'>
<ng-container *ngIf="event.imageURL">
<swiper-slide lazy="true" *ngFor="let imageURL of event.imageURL">
<img loading="lazy" [src]="imageURL">
</swiper-slide>
</ng-container>
</swiper-container>
<mat-card-content>
<p class="category">{{ event.category }}</p>
<p class="description">{{ event.description }}</p>
<div class="address-container">
<p class="time">{{ event.date | date: 'dd/MM/yyyy HH:mm' }}</p>
<p class="address">{{ event.address }}</p>
</div>
</mat-card-content>
```

```
<mat-card-footer class="actions">
<div class="button-container">
<button mat-fab class="join-button" [ngClass]="getButtonClass()"
(click)="toggleRegistration()">
<mat-icon>{{ isRegistered ? 'clear' : 'done' }}</mat-icon>
</button>
<button mat-fab class="people-button"
[matMenuTriggerFor]="partecipanti">
<mat-icon class="icon">people</mat-icon>
</button>
<div style="max-height: 300px; overflow-y: scroll;">
<mat-menu #partecipanti="matMenu">
<mat-list class="custom-list">
<mat-list-item *ngFor="let participant of event.participants"
style="display: block;">
{{ participant.name }}
</mat-list-item>
</mat-list>
</mat-menu>
</div>
<button mat-fab class="edit-button" (click)="editEvent()"
*ngIf="checkMyEvents()">
<mat-icon>edit</mat-icon>
</button>
<button mat-fab class="delete-button" (click)="deleteEvent()"
*ngIf="checkMyEvents()">
<mat-icon>delete</mat-icon>
</button>
</div>
</mat-card-footer>
</mat-card>
</section>
```

<a [routerLink]...>: Questo elemento <a> è utilizzato come collegamento per reindirizzare l'utente alla vista event-info relativa all'evento. Il reindirizzamento avviene quando l'utente fa clic sull'elemento.

<mat-card>: Questo elemento rappresenta una carta grafica Material Design che contiene i dettagli dell'evento. Il colore di sfondo della carta può essere personalizzato in base all'URL dell'immagine dell'evento.

<mat-card-header>: Questa sezione contiene il titolo dell'evento e la categoria dell'evento, con il colore di sfondo associato alla categoria.

<mat-card-content>: Questa sezione contiene la descrizione dell'evento.

<mat-card-footer>: Questa sezione contiene l'indirizzo dell'evento e la data, formattata come dd/MM/yyyy HH:mm.

```
import { CUSTOM_ELEMENTS_SCHEMA, Component, Inject, OnInit } from
'@angular/core';
import { MatDialog } from '@angular/material/dialog';
import { ActivatedRoute, Router } from '@angular/router';
import { AxiosService } from '../axios.service';
import { AxiosResponse } from 'axios';
import { RedirectService } from '../redirect.service';
import { MatDialogModule } from '@angular/material/dialog';
import { MatButtonModule } from '@angular/material/button';
import { MatListModule } from '@angular/material/list';
import { MatCardModule } from '@angular/material/card';
import { MatIconModule } from '@angular/material/icon';
import { CommonModule } from '@angular/common';
import { MAT_DIALOG_DATA } from '@angular/material/dialog';
import { MatMenu } from '@angular/material/menu';
import { MatMenuModule } from '@angular/material/menu';

@Component({
selector: 'app-event-info',
templateUrl: './event-info.component.html',
styleUrls: ['./event-info.component.css'],
standalone: true,
imports: [MatButtonModule, MatDialogModule, MatCardModule,
MatIconModule, CommonModule, MatMenuModule],
schemas: [CUSTOM_ELEMENTS_SCHEMA],
})
export class EventInfoComponent implements OnInit {
event: any;
registrationButtonLabel: string = 'Register';
isRegistered: boolean = false;
showPeopleList: boolean = false;

constructor(
public dialog: MatDialog,
private route: ActivatedRoute,
```

```
private axiosService: AxiosService,
private router: Router,
private service: RedirectService,
) {}

ngOnInit() {
this.axiosService.authenticate();

if (!this.event) {
// Se l'evento non è stato fornito, crea un evento di esempio
this.event = {
id: '',
title: '',
category: '',
description: '',
address: '',
date: new Date().toISOString(),
};
}
this.route.params.subscribe((params) => {
const id = +params['id'];
this.axiosService
.request('GET', `/api/event/findById/${id}`, {})
.then((response) => {
console.log('OK');
this.event = response.data;
console.log(this.event);
const imageURLs = this.event.imageURL;
const imageURLPromises = imageURLs.map((url: string) =>
this.loadURL(url)
);
Promise.all(imageURLPromises)
.then((urls) => {
this.event.imageURL = urls;
})
.catch((error) => {
console.error('Error loading image URLs:', error);
});
if (
this.isAlreadyLogged(
this.event.participants,
window.localStorage.getItem('userId')
) === true
```

```
) {
console.log(true);
this.isRegistered = true;
} else {
console.log(false);
this.isRegistered = false;
}
})
.catch((error) => {
console.log(error);
});
});
}

loadURL(endpoint: string): Promise<string> {
return new Promise((resolve, reject) => {
this.axiosService.customGet(endpoint).subscribe(
(response: AxiosResponse) => {
const reader = new FileReader();
reader.onload = () => {
resolve(reader.result as string);
};
reader.readAsDataURL(response.data);
},
(error) => {
console.error('Error retrieving the image:', error);
reject(error);
}
);
});
}

checkMyEvents() {
if (this.service.getRedirect() == '/my-events') {
return true;
}
return false;
}

isAlreadyLogged(participants: Array<any>, userId: string | null):
boolean {
if (!userId) {
return false;
```

```
}

for (const participant of participants) {
if (participant.id.toString() === userId) {
return true;
}
}

return false;
}

toggleRegistration() {
this.route.params.subscribe((params) => {
const eventId = +params['id'];
const userId = window.localStorage.getItem('userId');
this.isRegistered = !this.isRegistered;
this.registrationButtonLabel = this.isRegistered ? 'Unregister' :
'Register';

// Chiamata per aggiungere/rimuovere partecipante
const requestMethod = this.isRegistered ? 'POST' : 'DELETE';

this.axiosService
.request(requestMethod, `/api/event/${eventId}/register/${userId}`, {})
.then(() => {
console.log('Registration request succeeded');
// Altra chiamata per aggiornare la lista di partecipanti
this.updateParticipantsList(eventId);
})
.catch((error) => {
console.error('Registration request failed', error);
});
});
}

updateParticipantsList(eventId: number) {
this.axiosService
.request('GET', `/api/event/findById/${eventId}`, {})
.then((response) => {
console.log('Fetched updated participants list');
this.event.participants = response.data;
console.log(this.event.participants);
})
```

```
.catch((error) => {
console.error('Error fetching updated participants list', error);
});
}

getButtonClass() {
return this.isRegistered ? 'red-button' : 'join-button';
}

deleteEvent() {
// Qui puoi inserire la logica per la cancellazione dell'evento
// Ad esempio, puoi aprire un dialogo di conferma per la cancellazione
// e gestire l'eliminazione effettiva dell'evento
console.log('Deleting event...');

this.route.params.subscribe((params) => {
const eventId = +params['id'];
const userId = window.localStorage.getItem('userId');
this.axiosService
.request('DELETE', `/api/event/${eventId}/delete-event/${userId}`, {})
.then((response) => {
console.log('Event deleted');
this.router.navigate(['/event-board']);
})
.catch((error) => {
console.error('Error deleting event', error);
});
});
}

openDialog() {
console.log(this.event.participants);

// Passiamo i partecipanti al componente del dialogo PeopleList
const dialogRef = this.dialog.open(PeopleList, {
data: { participants: this.event.participants },
});

dialogRef.afterClosed().subscribe((result) => {
console.log(`Dialog result: ${result}`);
});
}
```

```
editEvent() {
this.route.params.subscribe((params) => {
const eventId = +params['id'];
this.router.navigate(['/event-edit', eventId]);
});
}
}
```

Il componente è decorato con il decorator @Component che definisce il selettore (selector), il modello (templateUrl) e il foglio di stile (styleUrls) associati al componente.

Il componente accetta un input event che rappresenta i dettagli dell'evento da visualizzare.

L'oggetto categoryColors definisce i colori associati alle diverse categorie di eventi. Questi colori vengono utilizzati per personalizzare lo sfondo del badge di categoria.

Il componente implementa l'interfaccia AfterViewInit e OnInit per eseguire azioni specifiche al caricamento del componente.

event-board:

```
<section>
<a class="filter-icon" mat-icon-button
routerLink="/event-board-filters">
<mat-icon>filter_list</mat-icon>
</a>
<swiper-container class="swiper" navigation="true" pagination="true"
pagination-clickable="true" slides-per-view= 'auto' free-mode="true"
space-between="30">
<swiper-slide class="slide" *ngFor="let event of events">
<app-event-card class="event-card" [event]="event"></app-event-card>
</swiper-slide>
<swiper-slide class="slide">
<app-event-blank></app-event-blank>
</swiper-slide>
</swiper-container>
</section>
```

<a class="filter-icon" mat-icon-button routerLink="/event-board-filters">: Questo elemento <a> è utilizzato come collegamento per reindirizzare l'utente alla vista "event-board-filters". Il reindirizzamento avviene quando l'utente fa clic sull'elemento. All'interno dell'elemento <a>, è presente un'icona Material Design "filter_list" rappresentata da <mat-icon>.

<swiper-container>: Questo elemento rappresenta un contenitore per la visualizzazione degli eventi tramite uno slider. Utilizza la libreria Swiper per creare un carosello di eventi.

<swiper-slide>: Questi elementi rappresentano le singole diapositive nello slider. Sono generati dinamicamente utilizzando *ngFor per iterare attraverso l'array di eventi (events) e mostrare un componente app-event-card per ciascun evento.

<app-event-card>: Questo componente viene utilizzato per visualizzare i dettagli di un evento. Riceve l'oggetto evento come input [event]="event".

<app-event-blank>: Questo componente rappresenta una diapositiva vuota all'interno dello slider. Può essere utilizzato per rappresentare una diapositiva vuota se non ci sono eventi da visualizzare.

```
import { Component, Input, OnInit } from '@angular/core';
import { AxiosService } from '../axios.service';
import { AxiosResponse } from 'axios';
import { MatDialog } from '@angular/material/dialog';
import { ActivatedRoute } from '@angular/router';
import { RedirectService } from '../redirect.service';

@Component({
selector: 'app-event-board',
templateUrl: './event-board.component.html',
styleUrls: ['./event-board.component.css'],
})
export class EventBoardComponent implements OnInit {
constructor(private axiosService: AxiosService, public dialog:
MatDialog, private route: ActivatedRoute, private redirectService:
RedirectService) {}
showFilterForm = false;

events: any[] = [ ];
filtro: any;

ngOnInit() {
```

```
console.log("Requesting events...");

this.axiosService.authenticate();
this.route.queryParams.subscribe((params) => {
if (params['filtro']) {
this.filtro = JSON.parse(params['filtro']);
console.log(this.filtro);
console.log(this.filtro.Titolo);
}
this.loadDataBasedOnFiltro();
});
console.log(this.filtro);
}

loadDataBasedOnFiltro() {
const filters = {
title: '',
place: '',
dateStart: '',
dateEnd: '',
category: ['EMPTY'],
typeEventPage: this.redirectService.getRedirect()
};

if (this.filtro) {
filters.title = this.filtro.Titolo || '';
filters.place = this.filtro.Luogo || '';
filters.dateStart = this.filtro.startDate || '';
filters.dateEnd = this.filtro.endDate || '';
filters.category = this.filtro.Categoria || [];
filters.typeEventPage = this.redirectService.getRedirect();
}

const userId = window.localStorage.getItem("userId");

console.log("CATEGORIA : " + filters.category);
console.log("eventPage : " + filters.typeEventPage);

this.axiosService.request(
"POST",
`/filter/${userId}`,
filters
).then(response => {
```

```
console.log("OK");
this.events = response.data;

const imageLoadingPromises = this.events.map(event =>
this.loadURL(event.imageURL));

Promise.all(imageLoadingPromises)
.then(imageURLs => {
imageURLs.forEach((imageURL, index) => {
this.events[index].imageURL = imageURL;
});

console.log(this.events);
})
.catch(error => {
console.error('Error loading images:', error);
});
})
.catch(error => {
console.log("Error");
});
}

loadURL(endpoint: string): Promise<string> {
return new Promise((resolve, reject) => {
this.axiosService.customGet(endpoint)
.subscribe((response: AxiosResponse) => {
const reader = new FileReader();
reader.onload = () => {
resolve(reader.result as string);
};
reader.readAsDataURL(response.data);
}, (error) => {
console.error('Error retrieving the image:', error);
reject(error);
});
});
}

toggleFilterForm() {
this.showFilterForm = !this.showFilterForm;
}
}
```

Il componente EventBoardComponent è un componente Angular che gestisce la visualizzazione di eventi nella vista dell'elenco eventi.
All'interno del costruttore del componente, vengono iniettati i servizi AxiosService, MatDialog, ActivatedRoute e RedirectService per l'accesso ai dati, la gestione dei dialoghi, il routing e la gestione del reindirizzamento.
showFilterForm è una variabile booleana utilizzata per controllare la visualizzazione del modulo di filtro nella vista.
events è un array che contiene gli eventi da visualizzare nella vista.
filtro è un oggetto utilizzato per memorizzare i parametri di filtro ricevuti attraverso i parametri dell'URL.
Il metodo **ngOnInit()** viene chiamato quando il componente viene inizializzato. In questo metodo, viene effettuata l'autenticazione, quindi vengono analizzati i parametri dell'URL per applicare eventuali filtri. Successivamente, il metodo **loadDataBasedOnFiltro()** viene chiamato per caricare gli eventi in base ai filtri applicati.
Il metodo **loadDataBasedOnFiltro()** gestisce il caricamento degli eventi in base ai filtri specificati e agli utenti connessi.
Il metodo **toggleFilterForm()** gestisce la visualizzazione o la chiusura del modulo di filtro quando l'utente fa clic sull'icona del filtro.

event-creation:

```
<section>
<div class="event">
<h2>Create a new event:</h2>
<form class="form" [formGroup]="eventForm" action=""
(ngSubmit)="onSubmit()">
<div class="inputbox">
<input matInput type="text" [(ngModel)]="newEvent.name" id="name"
formControlName="name"
placeholder="Insert a title" (focus)="fieldFocused = true"
(blur)="fieldFocused = false; fieldTouched = true">
<div class="alert" *ngIf="(fieldFocused || fieldTouched) &&
eventForm.get('name')?.hasError('required')">
Title missing!
```

```
</div>
</div>
<div class="inputbox">
<textarea class="description" matInput
[(ngModel)]="newEvent.description" id="description"
formControlName="description"
placeholder="Give your event a description"></textarea><br>
<div class="alert" *ngIf="(fieldFocused4 || fieldTouched4) &&
eventForm.get('description')?.hasError('required')">
Description missing!
</div>
</div>
<div class="inputbox">
<input matInput [(ngModel)]="newEvent.address" id="place"
formControlName="place"
placeholder="Via Marsala, 29H, 00185 Roma RM" (focus)="fieldFocused2 =
true" (blur)="fieldFocused2 = false; fieldTouched2 = true">
<div class="alert" *ngIf="(fieldFocused2 || fieldTouched2) &&
eventForm.get('place')?.hasError('required')">
Address missing!
</div>
<div class="alert" *ngIf="eventForm.get('place')?.hasError('pattern')">
Please specify civic, CAP and city! (Via Marsala, 29H, 00185 Rome RM)
</div>
</div>
<div class="inputbox">
<input class="date-time" type="datetime-local" id="dateTime"
[(ngModel)]="newEvent.dateTime" formControlName="dateTime">
<div class="alert"
*ngIf="eventForm.get('dateTime')?.hasError('dateRange')">
Please select a date between tomorrow and 1 year
</div>
</div>
<div class="inputbox">
<select class="category" id="category" [(ngModel)]="newEvent.category"
formControlName="category"
(focus)="fieldFocused3 = true" (blur)="fieldFocused3 = false;
fieldTouched3 = true">
<option value="" [selected]="!newEvent.category" disabled
selected>Select a category</option>
<option value="PARTY">Party</option>
<option value="GYM">Gym</option>
<option value="ROLEGAME">Role Game</option>
```

```
<option value="SPORT">Sport</option>
<option value="MEETING">Meeting</option>
<option value="CONFERENCE">Conference</option>
<option value="NETWORKING">Networking</option>
<option value="HOBBY">Hobby</option>
<option value="MUSIC">Music</option>
<option value="BUSINESS">Business</option>
<option value="FOOD">Food</option>
<option value="NIGHTLIFE">Nightlife</option>
<option value="HEALTH">Health</option>
<option value="HOLIDAYS">Holidays</option>
</select>
<div class="alert" *ngIf="(fieldFocused3 || fieldTouched3) &&
eventForm.get('category')?.hasError('required')">
Please select a category for the event
</div>
</div>
<div class="inputbox">
<input class="photo" type="file" name="images" accept="image/*"
(change)="onFileChange($event)" multiple formControlName="photo">
<div class="alert"
*ngIf="eventForm.get('photo')?.hasError('required')">
Photo missing!
</div>
</div>
<div class="actions">
<button mat-raised-button color="primary">Create</button>
</div>
</form>
</div>
</section>
```

Questo codice HTML rappresenta il form per la creazione di un nuovo evento. Il form contiene campi per inserire il titolo, la descrizione, l'indirizzo, la data e l'ora, la categoria e le immagini per l'evento.
Gli elementi <input>, <textarea>, <select>, e <button> sono associati a un modello con direttive [(ngModel)] e formControlName per gestire i dati del form.
Viene utilizzata la direttiva (focus) per rilevare il focus su un campo del form e (blur) per rilevare la perdita del focus. Vengono visualizzati messaggi di errore sotto i campi del form in caso di validazione fallita.

```
import { Component, OnInit } from '@angular/core';
import { AxiosService } from '../axios.service';
import { FormBuilder, FormGroup, Validators, AbstractControl, NgControl
} from '@angular/forms';
import { Router } from '@angular/router';

@Component({
selector: 'app-event-creation',
templateUrl: './event-creation.component.html',
styleUrls: ['./event-creation.component.css'],
})
export class EventCreationComponent implements OnInit{
newEvent: any = {};
imageFiles: File[] = []; // To store selected image files
fieldFocused: boolean = false;
fieldTouched: boolean = false;
fieldFocused2: boolean = false;
fieldTouched2: boolean = false;
fieldFocused3: boolean = false;
fieldTouched3: boolean = false;
fieldFocused4: boolean = false;
fieldTouched4: boolean = false;
eventForm: FormGroup;
customPattern = /.*?,.*?,.*\d.*\d.*\d.*\d./;

constructor(private axiosService: AxiosService, private formBuilder:
FormBuilder, private router: Router) {
this.eventForm = this.formBuilder.group({
name: [
'',
[
Validators.required
]
],
description: [
'',
[
Validators.required
]
```

```
],
place: [
'',
[
Validators.required,
Validators.pattern(this.customPattern)
]
],
dateTime: [
'',
[
Validators.required,
this.DateRangeValidator()
]
],
photo: [
'',
[
Validators.required
]
],
category: [
'',
[
Validators.required
]
]
})
}

ngOnInit(): void {
this.axiosService.authenticate();
}

DateRangeValidator(): Validators {
return (control: AbstractControl): { [key: string]: boolean } | null =>
{
if (control.value) {
const selectedDate = new Date(control.value);
const currentDate = new Date();
const minDate = new Date(currentDate);
minDate.setDate(currentDate.getDate()+ 1);
const maxDate = new Date(currentDate);
```

```
maxDate.setFullYear(currentDate.getFullYear() + 1);

if (selectedDate >= minDate && selectedDate <= maxDate) {
return null;
} else {
return { dateRange: true };
}
}
return null;
};
}

onSubmit(): void {

if (this.eventForm.valid) {
const userId = window.localStorage.getItem("userId");
const title = this.newEvent.name;
const description = this.newEvent.description;
const dateTimeLocal = this.newEvent.dateTime;
const place = this.newEvent.address;
const category = this.newEvent.category;

if (dateTimeLocal) {
const date = new Date(dateTimeLocal + 'Z');
const formattedDateTime = date.toISOString().substring(0, 23);
const endpoint = `/api/create-event/${userId}`;
const formData = new FormData();

for (const file of this.imageFiles) {
formData.append('photos', file, file.name);
}

formData.append('title', title);
formData.append('description', description);
formData.append('dateTime', formattedDateTime);
formData.append('place', place);
formData.append('category', category);

this.axiosService.requestMultipart(
"POST",
endpoint,
formData,
).then(response => {
```

```
console.log("redirecting to event-board...");
if (response.data === "Evento creato con successo") {
this.router.navigate(["/event-board"])
} else {
console.log(response.data);
}
});
console.log('Form submitted!');
} else {
console.error('Invalid date and time.');
}
}
}

onFileChange(event: any): void {
this.imageFiles = event.target.files;
}
}
```

Il componente EventCreationComponent è responsabile della creazione di un nuovo evento.

Viene utilizzato il modulo ReactiveFormsModule per creare un form reattivo e gestire la validazione dei campi.

Nel costruttore del componente, vengono definiti i campi del form con le relative regole di validazione utilizzando Validators.

Il metodo **DateRangeValidator()** restituisce una funzione di validazione personalizzata per garantire che la data e l'ora siano comprese tra domani e un anno dopo.

Il metodo **onSubmit()** viene chiamato quando il form viene inviato. Vengono effettuate le verifiche sulla validità del form e i dati vengono inviati al server se il form è valido.

Il metodo **onFileChange(event: any)** gestisce la selezione dei file immagine per l'evento.

filter-form:

```
<section>
<mat-form-field class="custom-input">
```

```
<mat-label>Title</mat-label>
<input matInput type="text" [(ngModel)]="filtro.Titolo" />
<button *ngIf="filtro.Titolo" matSuffix mat-icon-button
aria-label="Clear" (click)="filtro.Titolo = ''">
<mat-icon>close</mat-icon>
</button>
</mat-form-field>
<mat-form-field class="custom-input">
<mat-label>City</mat-label>
<input matInput type="text" [(ngModel)]="filtro.Luogo" />
<button *ngIf="filtro.Luogo" matSuffix mat-icon-button
aria-label="Clear" (click)="filtro.Luogo = ''">
<mat-icon>close</mat-icon>
</button>
</mat-form-field>
<mat-form-field class="custom-input">
<mat-form-field class="example-full-width">
<mat-label>From</mat-label>
<input matInput [matDatepicker]="fromPicker"
(dateInput)="filtro.startDate = $event.value">
<mat-hint>MM/DD/YYYY</mat-hint>
<mat-datepicker-toggle matIconSuffix [for]="fromPicker">
<mat-icon matDatepickerToggleIcon>keyboard_arrow_down</mat-icon>
</mat-datepicker-toggle>
<mat-datepicker #fromPicker></mat-datepicker>
</mat-form-field>
<mat-form-field class="example-full-width">
<mat-label>At</mat-label>
<input matInput [matDatepicker]="toPicker" (dateInput)="filtro.endDate
= $event.value">
<mat-hint>MM/DD/YYYY</mat-hint>
<mat-datepicker-toggle matIconSuffix [for]="toPicker">
<mat-icon matDatepickerToggleIcon>keyboard_arrow_down</mat-icon>
</mat-datepicker-toggle>
<mat-datepicker #toPicker></mat-datepicker>
</mat-form-field>
</mat-form-field>
<mat-form-field class="custom-input">
<mat-label>Category</mat-label>
<mat-select [formControl]="categoria" multiple
[(ngModel)]="filtro.Categoria">
<mat-option *ngFor="let categoria of categoriaList"
[value]="categoria">{{ categoria }}</mat-option>
```

```
</mat-select>
</mat-form-field>
<button mat-raised-button color="primary"
[routerLink]="['/event-board']" [queryParams]="{ filtro: filtro | json
}">Apply</button>
<p></p>
<p></p>
<button mat-raised-button color="accent"
routerLink="/event-board">Cancel</button>
</section>
```

Questo codice HTML rappresenta un modulo di filtro per eventi. Gli utenti possono inserire criteri di ricerca, come titolo, città, intervallo di date e categoria, e quindi applicare o annullare il filtro.
I campi di input per titolo e città sono associati a [(ngModel)] per mantenere i dati sincronizzati con il modello.
Sono presenti pulsanti di cancellazione per i campi di input del titolo e della città.
Sono inclusi campi di input per le date "From" e "At" con selettori di date associati. Gli utenti possono selezionare date specifiche per filtrare gli eventi in base all'intervallo di date desiderato.
È presente un menu a discesa per la selezione multipla delle categorie. Gli utenti possono selezionare una o più categorie per filtrare gli eventi.
Sono presenti pulsanti "Apply" e "Cancel" per applicare o annullare il filtro.

```
import { Component, OnInit } from '@angular/core';
import { FormControl } from '@angular/forms';
import { Router, ActivatedRoute } from '@angular/router';
import { RedirectService } from '../redirect.service';
import { AxiosService } from '../axios.service';

@Component({
selector: 'app-filter-form',
templateUrl: './filter-form.component.html',
styleUrls: ['./filter-form.component.css'],
})
export class FilterFormComponent implements OnInit {
categoria = new FormControl([]);
categoriaList: string[] = [
'PARTY', 'GYM', 'ROLEGAME', 'SPORT', 'MEETING', 'CONFERENCE',
```

```
'NETWORKING', 'HOBBY', 'MUSIC', 'BUSINESS', 'FOOD', 'NIGHTLIFE',
'HEALTH', 'HOLIDAYS'
];

filtro: any = {
Titolo: '',
Luogo: '',
startDate: null,
endDate: null,
Categoria: ['EMPTY'],
typeEventPage: this.redirectService.getRedirect()
};

private redirectTo: string;

constructor(
private router: Router,
private route: ActivatedRoute,
private redirectService: RedirectService,
private axiosService: AxiosService
) {
this.redirectTo = redirectService.getRedirect();
}

ngOnInit(): void {
this.axiosService.authenticate();
}

applicaFiltro() {
this.router.navigate(['/event-board'], {
relativeTo: this.route,
queryParams: { filtro: JSON.stringify(this.filtro) }
});
}
}
```

Il codice è un componente Angular che gestisce una form di filtro per gli eventi. Importa vari moduli e servizi, tra cui Component, FormControl, Router, ActivatedRoute, RedirectService, e AxiosService. Nel componente, vengono dichiarate proprietà per gestire input e filtro degli eventi. Nel costruttore, i servizi vengono iniettati e la variabile redirectTo viene inizializzata. Il metodo ngOnInit viene utilizzato per autenticare l'utente con il servizio AxiosService. Il metodo applicaFiltro permette di applicare il filtro e navigare alla pagina degli eventi. In

generale, il componente offre funzionalità di filtraggio e navigazione per gli eventi in un'applicazione Angular.

footer:

```
<footer class="d-flex flex-wrap justify-content-between
align-items-center py-3 my-4">
<span class="text-muted">&copy; {{currentYear}} Bravo Team</span>
</footer>
```

```
import { Component } from '@angular/core';

@Component({
selector: 'app-footer',
templateUrl: './footer.component.html',
styleUrls: ['./footer.component.css']
})
export class FooterComponent {
currentYear: number = new Date().getFullYear();
}
```

forgot-password:

```
<section>
<form [formGroup]="emailForm" class="container"
(ngSubmit)="onSubmit()">
<h2>Insert your email</h2>
<p>We will send a code to the following address in order to let you
reset the password</p>
<div class="inputbox">
<input type="email" id =email required placeholder=""
formControlName="email">
<div class="alert" *ngIf="emailForm.get('email')?.hasError('pattern')">
Please insert a valid email address
</div>
```

```
</div>
<button type="submit" [disabled]="emailForm.invalid"
(click)="ResetPassword()">
Send OTP code
</button>
<div class="inputbox error">
<div class="error-message" *ngIf="showError">
{{ errorMessage }}
</div>
</div>
</form>
</section>
```

Il modulo del form utilizza l'attributo [formGroup] per associare il modulo emailForm al form HTML.
Il campo email utilizza l'attributo formControlName="email" per collegare il campo dell'indirizzo email al controllo "email" nel modulo.
Viene mostrato un messaggio di errore se l'indirizzo email non è valido.
Il pulsante "Send OTP code" è abilitato solo se il modulo è valido.

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { AxiosService } from '../axios.service';
import { Router } from '@angular/router';

@Component({
selector: 'app-forgot-password',
templateUrl: './forgot-password.component.html',
styleUrls: ['./forgot-password.component.css']
})
export class ForgotPasswordComponent {
emailForm: FormGroup;
showError = false;
errorMessage = '';

constructor(private formBuilder: FormBuilder, private axiosService:
AxiosService, private router: Router) {
this.emailForm = this.formBuilder.group({
email: [
```

```
'',
[
Validators.required,
Validators.pattern(/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/)
]
]
});
}

ResetPassword() {
this.router.navigate(['/reset-password']);
}

onSubmit() {
console.log("submitted");
if (this.emailForm.valid) {
console.log("submitted");
const email = this.emailForm.get('email')?.value
this.axiosService.request2(
"POST",
"/api/auth/forgot-password",
{
email: email
}
).then(response => {
if (response.data === "Email not found") {
this.showError = true;
this.errorMessage = response.data;
} else {
this.showError = false;
this.errorMessage = '';
window.localStorage.setItem("emailForgotPassword", response.data)
this.router.navigate(['/reset-password']);
}
})
.catch(error => {
console.error('An error occurred during login:', error);
})

}
}
}
```

Il costruttore inizializza il modulo emailForm e inietta i servizi necessari.
Il metodo **ResetPassword()** gestisce il reindirizzamento dell'utente alla pagina di reset della password.
Il metodo **onSubmit()** viene chiamato quando il modulo viene inviato. Esegue una richiesta HTTP per il recupero della password e gestisce eventuali errori o esiti positivi.

register:

```
<section>
<div class="form-box">
<form [formGroup]="registerForm" action="" (ngSubmit)="onSubmit()">
<h2>Create a new profile:</h2>
<div class="inputbox">
<label for="firstName">First name</label>
<div class="input-container">
<input type="text" id="firstName" formControlName="firstName">
<div class="alert"
*ngIf="registerForm.get('firstName')?.hasError('pattern')">
Invalid first name!
</div>
</div>
</div>
<div class="inputbox">
<label for="lastName">Last name</label>
<div class="input-container">
<input type="text" id="lastName" formControlName="lastName">
<div class="alert"
*ngIf="registerForm.get('lastName')?.hasError('pattern')">
Invalid last name!
</div>
</div>
</div>
<div class="inputbox date">
<label for="dateOfBirth">Date of birth</label>
<div class="input-container">
<input type="date" id="date" formControlName="dateOfBirth">
<div class="alert"
*ngIf="registerForm.get('dateOfBirth')?.hasError('minAge') &&
!registerForm.get('dateOfBirth')?.hasError('required')">
You must be at least 18 to register!
</div>
</div>
```

```
</div>
<div class="inputbox">
<label for="email">Email</label>
<div class="input-container">
<input type="email" formControlName="email">
<div class="alert"
*ngIf="registerForm.get('email')?.hasError('pattern')">
Invalid email!
</div>
</div>
</div>
<div class="inputbox password">
<label for="password">Password</label>
<div class="input-container">
<div class="password-container">
<input type="{{ isPasswordVisible ? 'text' : 'password' }}"
id="password"
formControlName="password">
<mat-icon class="bi" (click)="togglePasswordVisibility()">
{{ isPasswordVisible ? 'visibility_off' : 'visibility' }}
</mat-icon>
</div>
<div class="alert"
*ngIf="(registerForm.get('password')?.hasError('pattern')) ||
(registerForm.get('password')?.hasError('minlength'))">
Make sure it's at least 8 characters including a capital letter, a
number and one special character.
</div>
</div>
</div>
<div class="inputbox password confirm-password">
<label for="confirm-password">Confirm password</label>
<div class="input-container">
<div class="password-container">
<input type="{{ isPassword2Visible ? 'text' : 'password' }}"
id="confirmPassword"
formControlName="confirmPassword">
<mat-icon class="bi" (click)="togglePasswordVisibility2()">
{{ isPassword2Visible ? 'visibility_off' : 'visibility' }}
</mat-icon>
</div>
<div class="alert"
```

```
*ngIf="registerForm.get('confirmPassword')?.value !==
registerForm.get('password')?.value">
Password does not match
</div>
</div>
</div>
<div class="inputbox upload">
<label for="fileInput" class="upload-label">Upload photo</label>
<div class="input-container">
<input type="file" #fileInput id="file" formControlName="photo"
accept="image/*">
<span id="fileName" class="file-name"></span>
<div class="photo-alert"
*ngIf="registerForm.get('photo')?.hasError('required')">
Photo missing!
</div>
</div>
</div>
<div class="inputbox privacy">
<label for="privacy" class="privacy-label">Accept terms and
conditions</label>
<div class="input-container">
<input class="checkbox" type="checkbox" formControlName="privacy">
</div>
</div>
<button type="submit"
[disabled]="!registerForm.valid">Register</button>
<div class="inputbox error">
<div class="error-message" *ngIf="showError">
{{ errorMessage }}
</div>
</div>
</form>
</div>
</section>
```

Il modulo del form utilizza l'attributo [formGroup] per associare il modulo registerForm al form HTML.
Ogni campo di input è associato a un controllo nel modulo tramite formControlName.
Vengono mostrati messaggi di errore personalizzati per i campi con regole di validazione specifiche.
Il pulsante "Register" è abilitato solo se il modulo è valido.

```
import { Component, ElementRef, ViewChild } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { AxiosService } from '../axios.service';
import { Router } from '@angular/router';
import axios from 'axios';

@Component({
selector: 'app-register',
templateUrl: './register.component.html',
styleUrls: ['./register.component.css'],
})
export class RegisterComponent {
showError = false;
errorMessage = '';
isPasswordVisible = false;
isPassword2Visible = false;
uploadedFileName = '';
@ViewChild('fileInput') fileInput: any;
registerForm: FormGroup;

constructor(private el: ElementRef, private formBuilder: FormBuilder,
private axiosService: AxiosService, private router: Router) {
this.registerForm = this.formBuilder.group({
firstName: [
'',
[
Validators.required,
Validators.pattern(/^[A-Za-z]+$/)
]
],
lastName: [
'',
[
Validators.required,
Validators.pattern(/^[A-Za-z]+$/)
]
],
dateOfBirth: [
'',
[
Validators.required,
this.ageValidator(18)
]
```

```
],
email: [
'',
[
Validators.required,
Validators.pattern(/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/)
]
],
password: [
'',
[
Validators.required,
Validators.minLength(8),
Validators.pattern(/^(?=.*[!@#$%^&*()_+[\]{}|~`<>?,.:;'"\\])(?=.*\d)(?=
.*[A-Z]).*$/)
]
],
confirmPassword: [
'',
[
Validators.required,
]
],
photo: [
'',
[
Validators.required,
]
],
privacy: [
false,
[
Validators.requiredTrue
]
],
});
}

togglePasswordVisibility() {
const input = this.el.nativeElement.querySelector('#password') as
HTMLInputElement;
this.isPasswordVisible = !this.isPasswordVisible;
input.type = this.isPasswordVisible ? 'text' : 'password';
```

```
}

togglePasswordVisibility2() {
const input = this.el.nativeElement.querySelector('#confirmPassword')
as HTMLInputElement;
this.isPassword2Visible = !this.isPassword2Visible;
input.type = this.isPassword2Visible ? 'text' : 'password';
}

ageValidator(minAge: number) {
return (control: FormGroup): { [key: string]: boolean } | null => {
const birthDate = new Date(control.value);
const today = new Date();
const age = today.getFullYear() - birthDate.getFullYear();

if (age < minAge) {
return { 'minAge': true };
}
return null;
};
};

onSubmit() {
const passwordValue = this.registerForm.get('password')?.value;
const confirmPasswordValue =
this.registerForm.get('confirmPassword')?.value;
if (this.registerForm.valid && passwordValue === confirmPasswordValue)
{
const formData = new FormData();
formData.append('firstName',
this.registerForm?.get('firstName')?.value);
formData.append('lastName', this.registerForm?.get('lastName')?.value);
formData.append('dateOfBirth',
this.registerForm?.get('dateOfBirth')?.value);
formData.append('email', this.registerForm?.get('email')?.value);
formData.append('password', this.registerForm?.get('password')?.value);
formData.append('confirmPassword',
this.registerForm?.get('confirmPassword')?.value);
formData.append('checkbox', this.registerForm?.get('privacy')?.value);

if (this.fileInput.nativeElement.files[0]) {
formData.append('photo', this.fileInput.nativeElement.files[0]);
} else {
```

```
const emptyBlob = new Blob([], { type: 'application/octet-stream' });
formData.append('photo', emptyBlob, 'empty.jpg');
}

if (this.registerForm?.get('dateOfBirth')?.value) {
formData.append('dateOfBirth',
this.registerForm?.get('dateOfBirth')?.value);
} else {
formData.append('dateOfBirth', '0000-00-00');
}

axios.post("/api/auth/signup", formData, {
headers: {
'Content-Type': 'multipart/form-data'
},
withCredentials: true,
})
.then(response => {
if (response.data === "Registered Succesfully") {
this.showError = false;
this.errorMessage = '';
this.router.navigate(['/login']);
} else {
this.showError = true;
this.errorMessage = response.data;
}
})
.catch(error => {
this.showError = true;
this.errorMessage = 'An unknown error occurred';
});
} else {
// Dati inseriti errati
}
}
}
```

Il costruttore inizializza il modulo registerForm con le regole di validazione per i campi.
Ci sono metodi per la gestione della visibilità delle password, la validazione dell'età e l'invio dei dati al server.
La logica di invio dei dati al server utilizza Axios per effettuare una richiesta HTTP.

login:

```
<section>
<div class="form-box">
<div class="form-value">
<form [formGroup]="loginForm" action="" (ngSubmit)="onSubmit()">
<h2>Login</h2>
<div class="inputbox">
<mat-icon class="icon">email</mat-icon>
<input type="email" formControlName="email" required placeholder="">
<label for="email">Email</label>
</div>
<div class="inputbox">
<mat-icon class="icon" (click)="togglePasswordVisibility()">
{{ isPasswordVisible ? 'visibility_off' : 'visibility' }}
</mat-icon>
<input type="{{ isPasswordVisible ? 'text' : 'password' }}"
formControlName="password" required
placeholder="">
<label for="password">Password</label>
</div>
<div class="forget">
<label for=""><a (click)="PasswordForgotten()">
Forgot Password?
</a></label>
</div>
<button type="submit" [disabled]="!loginForm.valid">Log in</button>
<div class="inputbox error">
<div class="error-message" *ngIf="showError">
{{ errorMessage }}
</div>
</div>
<div class="register">
<p>Don't have an account? <a (click)="Register()">Register here</a></p>
</div>
</form>
</div>
</div>
</section>
```

Il modulo del form utilizza l'attributo [formGroup] per associare il modulo loginForm al form HTML.
Sono presenti campi di input per l'email e la password.
È possibile attivare/disattivare la visibilità della password facendo clic sull'icona appropriata.
È possibile richiamare la funzione "Forgot Password?" per il ripristino della password.
Il pulsante "Log in" è abilitato solo se il modulo è valido.

```
import { Component, ElementRef } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { AxiosService } from '../axios.service';
import { Router } from '@angular/router';

@Component({
selector: 'app-login',
templateUrl: './login.component.html',
styleUrls: ['./login.component.css'],
})
export class LoginComponent {

showError = false;
errorMessage = '';
isPasswordVisible = false;
loginForm: FormGroup;

constructor(private formBuilder: FormBuilder, private el: ElementRef,
private axiosService: AxiosService, private router: Router){
this.loginForm = this.formBuilder.group({
email: [
'',
[
Validators.required,
Validators.pattern(/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/)
]
],
password: [
'',
[
Validators.required,
]
],
```

```
})
}

togglePasswordVisibility() {
const input = this.el.nativeElement.querySelector('#pwd') as
HTMLInputElement;
this.isPasswordVisible = !this.isPasswordVisible;
input.type = this.isPasswordVisible ? 'text' : 'password';
}

PasswordForgotten() {
this.router.navigate(['/forgot-password']);
}

Register() {
this.router.navigate(['/register']);
}

onSubmit() {
console.log("submitted");
if (this.loginForm.valid) {
console.log("sending login request");
this.axiosService.request2(
"POST",
"/api/auth/signin",
{
email: this.loginForm.get('email')?.value,
password: this.loginForm.get('password')?.value
}
).then(response => {
if (response.data.error !== null) {
this.showError = true;
this.errorMessage = response.data.error;
this.axiosService.request2(
"POST",
"/api/auth/signin-failure",
{
email: this.loginForm.get('email')?.value
})
} else {
console.log(response.data.email);
window.localStorage.setItem("email", response.data.email);
this.router.navigate(['/2FA-login']);
```

```
}
})
.catch(error => {
this.showError = true;
this.errorMessage = 'An error occurred during login:', error;
console.error('An error occurred during login:', error);
})

} else {
// Dati inseriti errati
}
}
}
```

Il costruttore inizializza il modulo loginForm con le regole di validazione per l'email e la password.
Ci sono metodi per gestire la visibilità della password, il ripristino della password, il reindirizzamento alla pagina di registrazione e l'invio dei dati al server.
La logica di invio dei dati al server utilizza Axios per effettuare una richiesta HTTP.

header:

```
<nav class="navbar">
<div class="title">
<img [src]="isMobile ? '/assets/titlemobile.png' : '/assets/title.png'"
alt="Image">
</div>
<button mat-icon-button (click)="navigateTo('/home')">
<mat-icon class="icon">home</mat-icon>
</button>
<button mat-icon-button (click)="updateRedirection('/event-board');
authenticate()">
<mat-icon class="icon">event</mat-icon>
</button>
<button mat-icon-button [matMenuTriggerFor]="notifiche"
(menuOpened)="SetNotificationAsRead(); resetNumeroNotifiche()">
<mat-icon class="icon" [hidden]="!this.getIsLogged()"
[matBadge]="numeroNotifiche" matBadgeSize="small"
```

```
matBadgeColor="primary" [matBadgeHidden]="numeroNotifiche ===
0">notifications</mat-icon>
</button>
<div style="max-height: 300px; overflow-y: scroll;">
<mat-menu #notifiche="matMenu">
<mat-list class="custom-list">
<button mat-menu-item *ngFor="let notification of notifications">
{{ notification.message }} - {{ notification.dateTime }}
</button>
</mat-list>
</mat-menu>
</div>
<button mat-icon-button [matMenuTriggerFor]="menu">
<mat-icon class="icon">person</mat-icon>
</button>
<mat-menu #menu="matMenu">
<a mat-menu-item (click)="navigateTo('/home')">Home</a>
<a mat-menu-item (click)="updateRedirection('/event-board');
authenticate()">Board</a>
<a mat-menu-item [hidden]="!this.getIsLogged()"
(click)="navigateToMyEvents()">My Events</a>
<a mat-menu-item [hidden]="!this.getIsLogged()"
(click)="navigateToRegisteredEvents()">Registered Event</a>
<a mat-menu-item [hidden]="this.getIsLogged()"
(click)="navigateTo('/login')">Sign in</a>
<a mat-menu-item [hidden]="this.getIsLogged()"
(click)="navigateTo('/register')">Register</a>
<a mat-menu-item [hidden]="!this.getIsLogged()"
(click)="navigateTo('/profile')">Profile</a>
<a mat-menu-item [hidden]="!this.getIsLogged()" (click)="logout()">Log
out</a>
</mat-menu>
</nav>
```

Il template mostra il titolo dell'applicazione, pulsanti per la navigazione tra le pagine e un menu di notifiche.
Il menu delle notifiche contiene una lista di notifiche.
Le icone dei pulsanti cambiano in base alle condizioni, come la presenza di notifiche non lette.

```
import { Component, Injectable, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { AxiosService } from '../axios.service';
import { RedirectService } from '../redirect.service';
import { HostListener } from '@angular/core';
import { SseService } from '../SseService';

@Injectable()
@Component({
selector: 'app-header',
templateUrl: './header.component.html',
styleUrls: ['./header.component.css']
})
export class HeaderComponent implements OnInit{

redirection = '';
numeroNotifiche: any = 0; // Initialize to 0
notifications: any[] = [
{
name: "boooooo"
},
]; // Store the notification list
imageName: string = '/assets/title.png'; // Imposta l'immagine
predefinita
isMobile: boolean = false;
profilePhoto: string = '';

constructor(
private router: Router,
private redirectService: RedirectService,
private axiosService: AxiosService,
private sseService: SseService
) {
this.sseService.getEvents().subscribe((eventData) => {
//aggiornare il numero di notifiche
console.log("arrived eventData");
for (const event of eventData) {
console.log(event.dateTime);
}
this.notifications = eventData;
console.log("pushed inside notification");
for (const notification of this.notifications) {
console.log(notification.dateTime);
```

```
}
this.numeroNotifiche = 0;
for (const notification of this.notifications) {
if (notification.read == false) {
this.numeroNotifiche++;
}
}
});
this.isMobile = window.innerWidth < 768;
}
ngOnInit(): void {
if (this.redirectService.getIsLogged() === true) {
const userId = window.localStorage.getItem("userId");
this.axiosService
.request('GET', `/show-notification/${userId}`, {})
.then((response) => {
console.log('test');
this.notifications = response.data;
console.log("pushed inside notification");
for (const notification of this.notifications) {
console.log(notification.dateTime);
}
this.numeroNotifiche = 0;
for (const notification of this.notifications) {
if (notification.read == false) {
this.numeroNotifiche++;
}
}
})
.catch((error) => {
console.log('error');
});
}
}


@HostListener('window:resize', ['$event'])
onResize(event: Event): void {
this.isMobile = window.innerWidth < 768;
}

logout() {
this.axiosService
```

```
.request2('POST', 'api/auth/logout', {})
.then((response) => {
this.redirectService.setIsLogged(false);
this.router.navigate(['/login']);
})
.catch((error) => {
this.redirectService.setIsLogged(false);
this.router.navigate(['/login']);
});
}

getIsLogged(): boolean {
// console.log(this.redirectService.getIsLogged());
return this.redirectService.getIsLogged();
}

updateRedirection(newRedirection: string) {
this.redirection = newRedirection;
}

authenticate() {
this.axiosService
.request('POST', 'api/authenticate', {})
.then((response) => {
console.log('test');
this.router.navigate([this.redirection]);
})
.catch((error) => {
console.log('error');
this.router.navigate(['/login']);
});
}

resetNumeroNotifiche() {
this.numeroNotifiche = 0;
}

navigateTo(route: string) {
this.redirectService.setRedirect(route);
this.router.navigate([route]);
}

navigateToMyEvents() {
```

```
this.redirectService.setRedirect('/my-events');
console.log(this.redirectService.getRedirect());
this.router.navigate(['/event-board']);
}

navigateToRegisteredEvents() {
this.redirectService.setRedirect('/registered-events');
console.log(this.redirectService.getRedirect());
this.router.navigate(['/event-board']);
}

SetNotificationAsRead() {
console.log("click on notification menu");

const userId = window.localStorage.getItem("userId");
this.axiosService.request(
"POST",
`/setNotificationRead/${userId}`,
{
notificationsToSet: this.numeroNotifiche
});
this.numeroNotifiche = 0;
}
}
```

Il costruttore inizializza il componente e gestisce le notifiche tramite il servizio SSE.
Ci sono metodi per la navigazione tra le pagine, l'autenticazione dell'utente e la gestione delle notifiche.
La variabile isMobile tiene traccia del layout mobile in base alla larghezza della finestra.

home:

```
<section>
<div class="overlay-container">
<div class="overlay-block" id="block1">
<i class="bi bi-globe2"></i>
<h3>Join events</h3>
<p>Search and partecipate in events created by other people</p>
</div>
```

```
<div class="overlay-block" id="block2">
<i class="bi bi-tools"></i>
<h3>Create your own events</h3>
<p>Create a new personalised public event for your friends or
strangers</p>
</div>
<div class="overlay-block" id="block3">
<i class="bi bi-person-arms-up"></i><i class="bi
bi-person-arms-up"></i><i class="bi bi-person-arms-up"></i>
<h3>Meet new people</h3>
<p>Meet up with the other partecipants and have fun with your old and
new friends</p>
</div>
</div>
</section>
```

```
import { Component } from '@angular/core';

@Component({
selector: 'app-home',
templateUrl: './home.component.html',
styleUrls: ['./home.component.css']
})
export class HomeComponent {

}
```

modify-events:

```
<section>
<div class="event">
<h2>Edit event</h2>
<form class="form" [formGroup]="eventForm" action=""
(ngSubmit)="onSubmit()">
```

```
<div class="inputbox">
<input matInput type="text" [(ngModel)]="newEvent.name" id="name"
formControlName="name"
placeholder="Insert a title" (focus)="fieldFocused = true"
(blur)="fieldFocused = false; fieldTouched = true">
<div class="alert" *ngIf="(fieldFocused || fieldTouched) &&
eventForm.get('name')?.hasError('required')">
Title missing!
</div>
</div>
<div class="inputbox">
<textarea class="description" matInput
[(ngModel)]="newEvent.description" id="description"
formControlName="description"
placeholder="Give your event a description"></textarea><br>
</div>
<div class="inputbox">
<input matInput [(ngModel)]="newEvent.address" id="place"
formControlName="place"(focus)="fieldFocused2 = true"
(blur)="fieldFocused2 = false; fieldTouched2 = true">
<div class="alert" *ngIf="(fieldFocused2 || fieldTouched2) &&
eventForm.get('place')?.hasError('required')">
Address missing!
</div>
<div class="alert" *ngIf="eventForm.get('place')?.hasError('pattern')">
Please specify civic, CAP and city! (Via Marsala, 29H, 00185 Rome RM)
</div>
</div>
<div class="inputbox">
<input class="date-time" type="datetime-local" id="dateTime"
[(ngModel)]="newEvent.dateTime" formControlName="dateTime">
<div class="alert"
*ngIf="eventForm.get('dateTime')?.hasError('dateRange')">
Please select a date within 1 year
</div>
</div>
<div class="inputbox">
<select class="category" id="category" [(ngModel)]="newEvent.category"
formControlName="category"
(focus)="fieldFocused3 = true" (blur)="fieldFocused3 = false;
fieldTouched3 = true">
<option value="" [selected]="!newEvent.category" disabled
selected>Select a category</option>
```

```
<option value="PARTY">Party</option>
<option value="GYM">Gym</option>
<option value="ROLEGAME">Role Game</option>
<option value="SPORT">Sport</option>
<option value="MEETING">Meeting</option>
<option value="CONFERENCE">Conference</option>
<option value="NETWORKING">Networking</option>
<option value="HOBBY">Hobby</option>
<option value="MUSIC">Music</option>
<option value="BUSINESS">Business</option>
<option value="FOOD">Food</option>
<option value="NIGHTLIFE">Nightlife</option>
<option value="HEALTH">Health</option>
<option value="HOLIDAYS">Holidays</option>
</select>
<div class="alert" *ngIf="(fieldFocused3 || fieldTouched3) &&
eventForm.get('category')?.hasError('required')">
Please select a category for the event
</div>
</div>
<div class="image-preview-container">
<div class="image-preview" *ngFor="let image of imageFiles; let i =
index">
<img [src]="image" alt="Image Preview">
<button mat-icon-button color="primary" (click)="removeImage(i)">
<mat-icon>close</mat-icon>
</button>
</div>
</div>

<div class="inputbox">
<input class="photo" type="file" name="images" accept="image/*"
(change)="onFileChange($event)" multiple formControlName="photo">
</div>
<div class="actions">
<button mat-raised-button color="primary"
(click)="onSubmit()">Submit</button>
</div>
</form>
</div>
</section>
```

Il template permette di modificare i dettagli di un evento, inclusi il nome, la descrizione, l'indirizzo, la data e l'immagine dell'evento.
È possibile caricare nuove immagini per l'evento.

```
import { Component } from '@angular/core';
import { AxiosService } from '../axios.service';
import { FormBuilder, FormGroup, Validators, AbstractControl } from
'@angular/forms';
import { ActivatedRoute } from '@angular/router';
import { AxiosResponse } from 'axios';

@Component({
selector: 'app-modify-events',
templateUrl: './modify-events.component.html',
styleUrls: ['./modify-events.component.css']
})
export class ModifyEventsComponent {
newEvent: any = {
name: '',
description: '',
address: '',
dateTime: '',
category: '',
image: ''
};
imageFiles: File[] = [];
fieldFocused: boolean = false;
fieldTouched: boolean = false;
fieldFocused2: boolean = false;
fieldTouched2: boolean = false;
fieldFocused3: boolean = false;
fieldTouched3: boolean = false;
eventForm: FormGroup;
customPattern = /.*?,.*?,.*\d.*\d.*\d.*\d./;

constructor(private axiosService: AxiosService, private formBuilder:
FormBuilder, private route: ActivatedRoute) {
this.eventForm = this.formBuilder.group({
name: [this.newEvent.name, [Validators.required]],
description: [this.newEvent.description, []],
```

```
place: [this.newEvent.address, [Validators.required,
Validators.pattern(this.customPattern)]],
dateTime: [this.newEvent.dateTime, [Validators.required,
this.DateRangeValidator()]],
photo: ['', [Validators.required]],
category: [this.newEvent.category, [Validators.required]]
});
}


event: any;

ngOnInit(): void {
this.axiosService.authenticate();

if (!this.event) {
// Se l'evento non è stato fornito, crea un evento di esempio
this.event = {
id: '',
title:'',
category: '',
description: '',
address: '',
date: new Date().toISOString(),
// image:
};
this.route.params.subscribe(params => {
const id = +params['id'];
this.axiosService.request("GET", `/api/event/findById/${id}`, {})
.then(response => {
console.log("OK");
this.event = response.data;
console.log(this.event);
const imageURLs = this.event.imageURL;
const imageURLPromises = imageURLs.map((url: string) =>
this.loadURL(url));
Promise.all(imageURLPromises)
.then(urls => {
this.event.imageURL = urls;
})
.catch(error => {
console.error('Error loading image URLs:', error);
});
```

```
})
})
}
}

loadURL(endpoint: string): Promise<string> {
return new Promise((resolve, reject) => {
this.axiosService.customGet(endpoint)
.subscribe((response: AxiosResponse) => {
const reader = new FileReader();
reader.onload = () => {
resolve(reader.result as string);
};
reader.readAsDataURL(response.data);
}, (error) => {
console.error('Error retrieving the image:', error);
reject(error);
});
});
}

DateRangeValidator(): Validators {
return (control: AbstractControl): { [key: string]: boolean } | null =>
{
if (control.value) {
const selectedDate = new Date(control.value);
const currentDate = new Date();
const maxDate = new Date();
maxDate.setFullYear(maxDate.getFullYear() + 1);

if (selectedDate < currentDate || selectedDate > maxDate) {
return { dateRange: true };
}
}
return null;
};
}

onSubmit(): void {
if (this.eventForm.valid) {
this.route.params.subscribe(params => {
const eventId = +params['id'];
const userId = window.localStorage.getItem("userId");
```

```
const title = this.newEvent.name;
const description = this.newEvent.description;
const dateTimeLocal = this.newEvent.dateTime;
const place = this.newEvent.location;
const category = this.newEvent.category;

if (dateTimeLocal) {
const date = new Date(dateTimeLocal + 'Z');
const formattedDateTime = date.toISOString().substring(0, 23);
const endpoint = `/api/event/${eventId}/modify-event/${userId}`;
const formData = new FormData();

for (const file of this.imageFiles) {
formData.append('photos', file, file.name);
}

formData.append('title', title);
formData.append('description', description);
formData.append('dateTime', formattedDateTime);
formData.append('place', place);
formData.append('category', category);

this.axiosService.requestMultipart(
"PUT",
endpoint,
formData
);
console.log('Form submitted!');
} else {
console.error('Invalid date and time.');
}
})
}
}

onFileChange(event: any): void {
this.imageFiles = event.target.files;
}

removeImage(index: number): void {
this.imageFiles.splice(index, 1);
}
}
```

Il costruttore inizializza il componente e il form per la modifica dell'evento.
Il metodo **DateRangeValidator()** è utilizzato per validare la data dell'evento.
Il metodo **onSubmit()** gestisce l'invio delle modifiche.
Il metodo **onFileChange()** cattura le immagini caricate dall'utente.
Il metodo **removeImage()** permette di rimuovere un'immagine dalla lista delle immagini.

profile:

```
<section>
<mat-card class="profile">
<mat-card-header class="header">
<mat-card-title-group>
<mat-card-title class="title">Profile info:</mat-card-title>
<img mat-card-md-image class="photo" [src]="profilePhoto">
</mat-card-title-group>
</mat-card-header>
<mat-card-content class="info">
<form [formGroup]="profileForm" action="" (ngSubmit)="SaveChanges()">
<div class="inputbox">
<input type="text" id="firstname" [(ngModel)]="firstname"
[readonly]="!modifyPermit"
[ngClass]="{'input-modify': modifyPermit}" formControlName="firstname">
<div class="alert"
*ngIf="profileForm.get('firstname')?.hasError('pattern')">
Invalid first name!
</div><br>
</div>
<div class="inputbox">
<input type="text" id="lastname" [(ngModel)]="lastname"
[readonly]="!modifyPermit"
[ngClass]="{'input-modify': modifyPermit}" formControlName="lastname">
<div class="alert"
*ngIf="profileForm.get('lastname')?.hasError('pattern')">
Invalid last name!
</div><br>
</div>
<div class="inputbox">
<input type="email" id="email" [ngModel]="oldEmail"
[readonly]="!modifyPermit" [ngClass]="{'input-modify': modifyPermit}"
formControlName="email" (ngModelChange)="EmailChange($event)">
<div class="alert"
*ngIf="profileForm.get('email')?.hasError('pattern')">
Invalid email!
```

```
</div><br>
</div>
<div class="inputbox">
<input type="text" id="date" value="{{date}}" [(ngModel)]="date"
readonly>
</div>
<div class="photo-alert" [hidden]="!modifyPermit">
Upload a new profile photo:
</div>
<input type="file" class="file" #fileInput [hidden]="!modifyPermit"
accept="image/*" (change)="UploadNewPhoto($event)">
</form>
</mat-card-content>
<div class="button">
<button mat-fab class="modify-button" (click)="ModifyProfile()"
[style.display]="showButton ? 'flex' : 'none'">
<mat-icon>edit</mat-icon>
</button>
<button type="submit" [disabled]="!profileForm.valid"
mat-fab class="confirm-button" [style.display]="!showButton ? 'flex' :
'none'"
(click)="SaveChanges()">
<mat-icon>done</mat-icon>
</button>
<button mat-fab class="delete-button" (click)="ReloadPage()"
[style.display]="!showButton ? 'flex' : 'none'">
<mat-icon>clear</mat-icon>
</button>
</div>
</mat-card>
</section>
```

Il template visualizza le informazioni sul profilo, inclusi il nome, il cognome, l'email e la foto del profilo.
L'utente può modificare alcune informazioni del profilo e caricare una nuova foto.

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Router } from '@angular/router';
```

```
import { AxiosService } from '../axios.service';
import { AxiosResponse } from 'axios';

@Component({
selector: 'app-profile',
templateUrl: './profile.component.html',
styleUrls: ['./profile.component.css']
})
export class ProfileComponent implements OnInit{
profileForm: FormGroup;

constructor(private formBuilder: FormBuilder, private router: Router,
private axiosService: AxiosService) {
this.profileForm = this.formBuilder.group({
firstname: [
'',
[
Validators.required,
Validators.pattern(/^[A-Za-z]+$/)
]
],
lastname: [
'',
[
Validators.required,
Validators.pattern(/^[A-Za-z]+$/)
]
],
email: [
'',
[
Validators.required,
Validators.pattern(/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/)
],
]
});
}

firstname: string = '';
lastname: string = '';
oldEmail: string = '';
newEmail: string = '';
date: string = '';
```

```
profilePhoto: string = '';

ngOnInit(): void {

const userId = window.localStorage.getItem("userId");

this.axiosService.request(
"GET",
`/api/getProfileInfo/${userId}`,
{}
).then(response => {
this.firstname = response.data.firstName;
this.lastname = response.data.lastName;
this.oldEmail = response.data.email;
this.newEmail = response.data.email;
this.date = response.data.date;
this.loadURL(response.data.imageUrl)
.then((result) => {
this.profilePhoto = result;
console.log(this.profilePhoto);
})
.catch((error) => {
console.error('Error loading profile photo:', error);
});
})
throw new Error('Method not implemented.');
}



modifyPermit: boolean = false;
showButton: boolean = true;

ModifyProfile() {
this.modifyPermit = true;
this.showButton = false;
}

EmailChange(tmp: string) {
this.newEmail = tmp;
}

SaveChanges() {
```

```
if (this.newEmail != this.oldEmail) {
this.router.navigate(['/2FA-login']);
}
else {
// salva firstname e lastname nel database
window.location.reload();
}
}

ReloadPage() {
window.location.reload();
}

UploadNewPhoto(event: any) {
const file = event.target.files[0]; // Ottieni il file selezionato
dall'input
if (file) {
const reader = new FileReader();
reader.onload = (e: any) => {
this.profilePhoto = e.target.result; // Imposta l'immagine visualizzata
sulla pagina con i dati del file caricato
};
reader.readAsDataURL(file);
}
}

loadURL(endpoint: string): Promise<string> {
return new Promise((resolve, reject) => {
this.axiosService.customGet(endpoint)
.subscribe((response: AxiosResponse) => {
const reader = new FileReader();
reader.onload = () => {
resolve(reader.result as string);
};
reader.readAsDataURL(response.data);
}, (error) => {
console.error('Error retrieving the image:', error);
reject(error);
});
});
}
}
```

Il costruttore inizializza il componente e il form per il profilo utente.
Il metodo **ngOnInit()** recupera le informazioni del profilo dell'utente.
Il metodo **ModifyProfile()** abilita la modifica del profilo.
Il metodo **SaveChanges()** consente di salvare le modifiche apportate al profilo.
Il metodo **UploadNewPhoto()** consente di caricare una nuova foto del profilo.

reset-password:

```
<section>
<form [formGroup]="resetForm" class="container" (ngSubmit)="onSubmit()">
<h2>Insert the OTP code found in the email you just received</h2>
<div class="inputbox otp">
<input type="text" id="otp" name="otp" required placeholder="" formControlName="otp">
<label for="otp">OTP code</label>
<div class="alert" *ngIf="resetForm.get('otp')?.hasError('pattern')">
Invalid OTP
</div>
</div>
<div class="inputbox">
<div class="password-container">
<input type="{{ isNewPasswordVisible ? 'text' : 'password' }}" id="newPassword" required
placeholder="" formControlName="newPassword">
<span class="bi" [ngClass]="{'bi-eye': !isNewPasswordVisible, 'bi-eye-slash':
isNewPasswordVisible}" (click)="toggleNewPasswordVisibility()"></span>
</div>
<label for="newPassword">New password</label>
<div class="alert" *ngIf="(resetForm.get('newPassword')?.hasError('pattern')) ||
(resetForm.get('newPassword')?.hasError('minlength'))">
Password must contain at least one capital letter. <br>
Password must be long at least 8 characters. <br>
Password must contain at least one number and one special character.
</div>
</div>
<div class="inputbox">
<div class="password-container">
<input type="{{ isConfirmPasswordVisible ? 'text' : 'password' }}" id="confirmPassword"
required placeholder="" formControlName="confirmPassword">
<span class="bi" [ngClass]="{'bi-eye': !isConfirmPasswordVisible, 'bi-eye-slash':
isConfirmPasswordVisible}" (click)="toggleConfirmPasswordVisibility()"></span>
</div>
<label for="confirmPassword">Confirm password</label>
```

```
<div class="alert" *ngIf="resetForm.get('confirmPassword')?.value !==
resetForm.get('newPassword')?.value">
Password does not match
</div>
</div>
<div class="resend-email">
<a [class.email.sent]="emailSent" (click)="resendEmail()"
[class.clicked]="emailSent">Resend email</a>
<label *ngIf="emailSent">Email sent</label>
</div>
<button type="submit" [disabled]="resetForm.invalid">Reset password</button>
<div class="inputbox error">
<div class="error-message" *ngIf="showError">
{{ errorMessage }}
</div>
</div>
</form>
</section>
```

Il template permette di inserire il codice OTP e la nuova password per il reset della password.
È possibile nascondere o mostrare la password usando l'icone "eye".
È possibile rispedire l'email con il codice OTP.
Un messaggio di errore viene mostrato se ci sono errori nel form.

```
import { Component, ElementRef } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { AxiosService } from '../axios.service';

@Component({
selector: 'app-reset-password',
templateUrl: './reset-password.component.html',
styleUrls: ['./reset-password.component.css']
})
export class ResetPasswordComponent {
resetForm: FormGroup;
showError = false;
errorMessage = '';
```

```
constructor(private formBuilder: FormBuilder, private el: ElementRef,
private router: Router, private axiosService: AxiosService) {
this.resetForm = this.formBuilder.group({
otp: [
'',
[
Validators.required,
Validators.pattern(/^\d{4,6}$/)
]
],
newPassword: [
'',
[
Validators.required,
Validators.minLength(8),
Validators.pattern(/^(?=.*[!@#$%^&*()_+[\]{}|~`<>?,.:;'"\\])(?=.*\d)(?=
.*[A-Z]).*$/)
]
],
confirmPassword: ['', Validators.required]
});
}

isNewPasswordVisible = false;
isConfirmPasswordVisible = false;

toggleNewPasswordVisibility() {
const input = this.el.nativeElement.querySelector('#password') as
HTMLInputElement;
this.isNewPasswordVisible = !this.isNewPasswordVisible;
input.type = this.isNewPasswordVisible ? 'text' : 'password';
}

toggleConfirmPasswordVisibility() {
const input = this.el.nativeElement.querySelector('#confirm-password')
as HTMLInputElement;
this.isConfirmPasswordVisible = !this.isConfirmPasswordVisible;
input.type = this.isConfirmPasswordVisible ? 'text' : 'password';
}

public emailSent: boolean = false;

resendEmail() {
```

```
this.axiosService.request2(
"POST",
"/api/auth/refresh-2FA",
{
email: window.localStorage.getItem("emailForgotPassword")
})
this.emailSent = true;
}

onSubmit() {
console.log("submitted");

if (this.resetForm.valid) {
const otpValue = this.resetForm.get('otp')?.value;
const newPasswordValue = this.resetForm.get('newPassword')?.value;
const confirmPasswordValue =
this.resetForm.get('confirmPassword')?.value;

if (newPasswordValue === confirmPasswordValue)
{
this.axiosService.request2(
"POST",
"/api/auth/reset-password",
{
otp: otpValue,
password: newPasswordValue,
confirmPassword: confirmPasswordValue
}
).then(response => {
if (response.data !== "Reset password done correctly") {
this.showError = true;
this.errorMessage = response.data;
} else {
this.showError = false;
this.errorMessage = '';
this.router.navigate(['/login']);
}
})
.catch(error => {
console.error('An error occurred during login:', error);
})
} else {
}
```

```
}
}
}
```

Il costruttore inizializza il componente e il form per il reset della password.
I metodi **toggleNewPasswordVisibility()** e **toggleConfirmPasswordVisibility()** consentono di nascondere o mostrare la password.
Il metodo **resendEmail()** permette di rispedire l'email con il codice OTP.
Il metodo **onSubmit()** gestisce il reset della password e visualizza eventuali messaggi di errore.

two-fa-login:

```
<section>
<form [formGroup]="twoFAForm" class="container"
(ngSubmit)="onSubmit()">
<h2>Insert the OTP code found in the email you just received</h2>
<div class="inputbox otp">
<input type="text" id="otp" name="otp" required placeholder=""
formControlName="otp">
<label for="otp">OTP code</label>
<div class="alert" *ngIf="twoFAForm.get('otp')?.hasError('pattern')">
Invalid OTP
</div>
<div class="inputbox error">
<div class="error-message" *ngIf="showError">
{{ errorMessage }}
</div>
</div>
</div>
<div class="resend-email">
<a [class.email.sent]="emailSent" (click)="resendEmail()"
[class.clicked]="emailSent">Resend email</a>
<label *ngIf="emailSent">Email sent</label>
</div>
<button type="submit" [disabled]="twoFAForm.invalid">Sign in</button>
</form>
</section>
```

Il template consente di inserire il codice OTP ricevuto via email.
È possibile rispedire l'email con il codice OTP.

Un messaggio di errore viene mostrato se l'OTP è invalido o se si verificano errori durante l'autenticazione.

```
import { Component, ElementRef } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { AxiosService } from '../axios.service';
import { Router } from '@angular/router';
import { RedirectService } from '../redirect.service';

@Component({
selector: 'app-two-fa-login',
templateUrl: './two-fa-login.component.html',
styleUrls: ['./two-fa-login.component.css']
})
export class TwoFALoginComponent {
twoFAForm: FormGroup;
showError = false;
errorMessage = '';

constructor(private formBuilder: FormBuilder, private el: ElementRef,
private axiosService: AxiosService, private router: Router, private
redirectService: RedirectService) {
this.twoFAForm = this.formBuilder.group({
otp: [
'',
[
Validators.required,
Validators.pattern(/^\d{4,6}$/)
]
]
});
}

public emailSent: boolean = false;

resendEmail() {
console.log("resending email");
this.axiosService.request2(
"POST",
"/api/auth/refresh-2FA",
{
```

```
email: window.localStorage.getItem("email")
})
this.emailSent = true;
}
onSubmit() {
console.log("submitted otp");
if (this.twoFAForm.valid) {
console.log("submitted otp");
const otpValue = this.twoFAForm.get('otp')?.value;
this.axiosService.request2(
"POST",
"/api/auth/2FA",
{
otp: otpValue
}
).then(response => {
if (response.data.error !== null) {
this.showError = true;
this.errorMessage = response.data.error;
} else {
this.showError = false;
this.errorMessage = '';
console.log(response.data.access_token);
window.localStorage.setItem("expiration_date",
response.data.expiration_date);
window.localStorage.setItem("userId", response.data.userId);
console.log(response.data.expiration_date);
this.redirectService.setIsLogged(true);
this.router.navigate(['/home']);
}
})
.catch(error => {
console.error('An error occurred during login:', error);
})
} else {
// newPassword e confirmPassword non corrispondono
}
}
}
```

Il costruttore inizializza il componente e il form per l'autenticazione a due fattori (2FA).
Il metodo resendEmail() permette di rispedire l'email con il codice OTP.

Il metodo onSubmit() gestisce l'autenticazione a due fattori (2FA) e visualizza eventuali messaggi di errore.