



UNIVERSITÀ  
DEGLI STUDI DI BARI  
ALDO MORO

# Secure Web Application

Dipartimento di Informatica

*Corso di Laurea Magistrale in “Sicurezza Informatica”*

**Sicurezza nelle Applicazioni [063673]**

Prof. Donato Malerba

Prof. Paolo Mignone

Studente

**Gabriel Cellammare**

Matricola: **807350**

Email: [g.cellammare1@studenti.uniba.it](mailto:g.cellammare1@studenti.uniba.it)

# Sommario

## Sommario

Sommario.....	2
Obiettivo del progetto .....	5
Progettazione del sistema.....	5
Implementazione del sistema.....	5
Gestione del database.....	7
Analisi Statica.....	8
Lifetime dei dati sensibili .....	8
Cookie .....	10
Funzionamento meccanismo “Ricordami” .....	12
Reindirizzamento automatico tramite Cookie permanenti .....	15
Sessione HTTPS/TLS.....	18
JDBC.....	23
Prepared Statements (PS) e la loro importanza .....	23
Crittografia.....	26
Gestione File .....	28
Immagine del profilo .....	29
Proposta progettuale .....	30
Visualizzazione della proposta progettuale.....	33
Caricamento di una proposta progettuale.....	36
Gestione Password.....	38
Applicazione Salt e Hash.....	38
Fase di registrazione .....	40
Fase di Login .....	42
Minimizzazione scope variabili.....	42
Accessibilità delle classi.....	43
Feedback dei risultati all’utente .....	46
Implementazione feedback visivi.....	48
Thread Safety .....	49
Analisi Dinamica .....	52
Test d’uso.....	52
Caso d’uso I: Registrazione effettuata correttamente .....	52
Caso d’uso II: Login effettuato correttamente senza meccanismo Remember me.....	53
Caso d’uso III: Login effettuato correttamente con meccanismo Remember me .....	54
Caso d’uso IV: Logout manuale effettuato correttamente.....	55

Caso d'uso V: Logout automatico effettuato correttamente .....	56
Caso d'uso VI: Proposta caricata in maniera corretta .....	56
Caso d'uso VII: L'utente visualizza tutte le proposte progettuali.....	57
Caso d'uso VIII: L'utente visualizza una proposta progettuale specifica .....	58
Caso d'uso IX: L'utente non loggato visita la pagina di login con cookie precedentemente registrato senza una sessione inizializzata .....	59
Caso d'uso X: L'utente loggato tenta di effettuare nuovamente il login dopo aver visitato l'Homepage	59
Test d'abuso .....	60
Caso d'uso XI: Registrazione con e-mail già in uso .....	60
Caso d'uso XII: Registrazione con password debole .....	61
Caso d'uso XIII: Registrazione con immagine profilo avente estensione non valida.....	61
Caso d'uso XIV: Registrazione con immagine profilo avente estensione valida ma formato non coerente .....	62
Caso d'uso XV: Login con password errata .....	63
Caso d'uso XVI: Login con e-mail errata .....	64
Caso d'uso XVII: L'utente privo di cookie cerca di visitare la pagina protetta senza aver effettuato il login .....	65
Caso d'uso XVIII: L'utente con cookie registrato ma scaduto cerca di visitare la pagina protetta senza aver effettuato il login .....	65
Caso d'uso XIX: Caricamento della proposta con script incorporato .....	65
Caso d'uso XX: Caricamento di un file con estensione diversa da .txt.....	67
Glossario .....	68



# Obiettivo del progetto

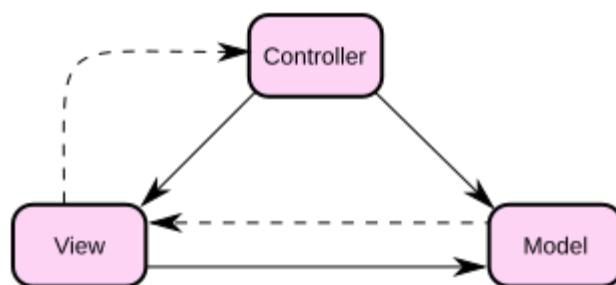
Il progetto “Secure Web Application” richiedeva le seguenti funzionalità:

- Registrazione con immagine di profilo
  - L'applicazione consente di registrarsi con e-mail, password, retype password, foto profilo (*ad esempio .png, jpeg...*)
- Login
  - L'applicazione consente di chiedere all'utente di voler effettuare il login con o senza il meccanismo dei Cookie
- Caricamento di una proposta progettuale in formato .txt
- Visualizzazione delle altre proposte progettuali all'interno della sessione https e/o della validità dei Cookie
- Logout
  - Invalida la sessione Http e i Cookie
- L'applicazione è inoltre in grado di fornire feedback all'utente dipendenti dal tipo di esito delle richieste (Registrazione OK, Registrazione KO, Immagine non valida, Errore login, Sessione scaduta...)

## Progettazione del sistema

L'architettura adottata è stata quella del **MVC** (*Model-View-Controller*) in particolare nell'ambito della programmazione orientata agli oggetti e in applicazioni web, è in grado di separare la logica di presentazione dei dati dalla logica di business. Questo pattern si posiziona nel livello logico o di business e di presentazione in una architettura multi-tier.

Il **modello** all'interno dell'architettura MVC rappresenta il nucleo logico e gestisce i dati e le regole dell'applicazione, separandoli dall'interfaccia utente. La **vista** è la rappresentazione visuale delle informazioni gestite dal modello, e possono esserci più viste per gli stessi dati, come grafici o tabelle. Il **controller** agisce come intermediario, ricevendo l'input dall'utente e convertendolo in comandi per aggiornare il modello o la vista, orchestrando l'interazione tra le diverse componenti per mantenere la separazione delle responsabilità.

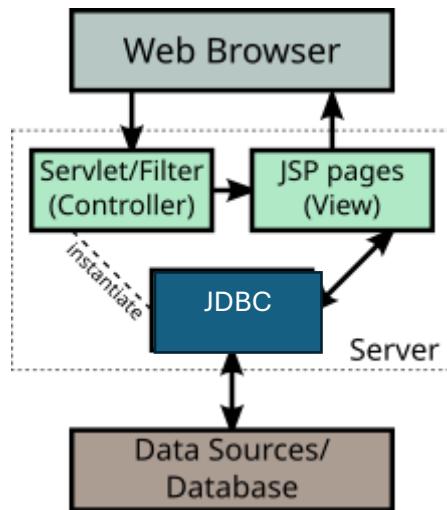


## Implementazione del sistema

Come linguaggio di back-end per la logica dell'applicazione è stato utilizzato **Java**, mentre per il front-end si è fatto uso di pagine **JSP**, che consentono di generare contenuti **HTML** dinamici. Una pagina **JSP** (*Jakarta Server Page o Java Server Page*) è una collezione di tecnologie (*HTML, XML...*) che aiuta gli sviluppatori software a creare pagine web generate dinamicamente. È simile a **PHP** e **ASP** ma usa il linguaggio Java. Per effettuare il **deploy** di una JSP è necessario un servlet container come **Apache Tomcat**. Il database utilizzato per la gestione dei dati è **MySQL**, un sistema di gestione di database relazionale.

Java interagisce con MySQL tramite **JDBC** (*Java Database Connectivity*), permettendo l'archiviazione e il recupero delle informazioni necessarie all'applicazione web.

Il browser si occuperà quindi di renderizzare la **View**, il **controller** tramite delle **Servlet**<sup>1</sup> gestite con Tecnologia **Java** si interfaccerà con il **Model**, che istanzierà le connessioni necessarie per comunicare con il **Database**.



Nello specifico, all'interno delle pagine **JSP** è stata utilizzata la tecnologia **Javascript** e **Ajax**. Un'applicazione web utilizza **AJAX** per inviare una richiesta **GET** asincrona al server ottenendo dati, senza dover ricaricare l'intera pagina. Ad esempio, un'applicazione web potrebbe utilizzare **AJAX** per aggiornare dinamicamente una sezione della pagina web. **Una chiamata AJAX** è una tecnologia che permette di **ottenere dati da un server senza dover ricaricare la pagina**. È l'acronimo di “*Asynchronous JavaScript and XML*”, ed è alla base dello sviluppo web moderno. **AJAX** può essere utilizzato attraverso **JQuery** - *nota libreria di Javascript* - utilizzando una sintassi più concisa.

```

$.ajax({
    url: 'UploadProposalServlet', // URL dell'API o del server per ottenere le proposte
    method: 'GET',
    dataType: 'json',
    success: function (data) {
        console.log("JSON ricevuto dal server:", data); // Log del JSON ricevuto per il debug

        var proposalList = $('#proposalBannerList'); // Seleziona l'elemento della lista delle proposte
        proposalList.empty(); // Svuota la lista esistente per evitare duplicati

        if (!scrittaAggiunta) {
            proposalList.append('<p><strong>Lista proposte progettuali</strong></p>'); // Aggiunge un titolo un
            scrittaAggiunta = false;
        }

        $.each(data, function (index, proposal) {
            console.log("Proposta corrente:", proposal); // Log della proposta corrente per il debug
        });
    }
});
```

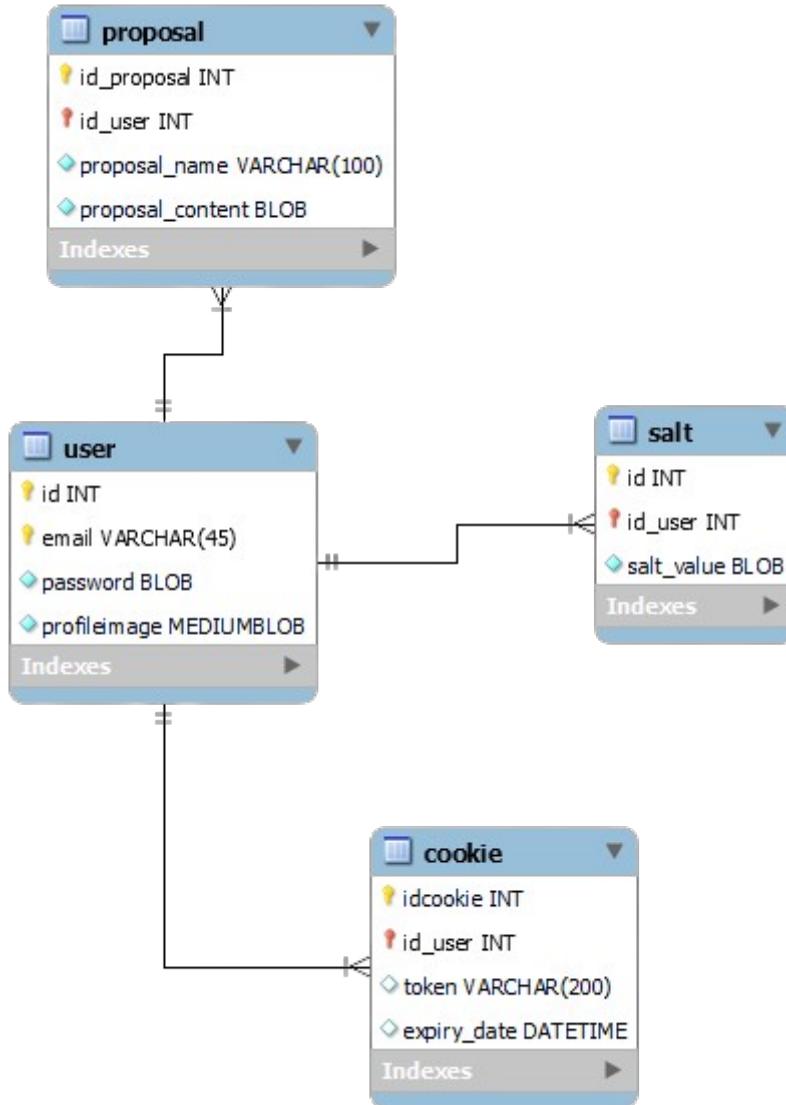
Esempio di una chiamata **AJAX**

---

<sup>1</sup> Una **servlet** è una classe Java che implementa i metodi GET e POST per la comunicazione http tramite i metodi *void doGet e doPost*

## Gestione del database

Dopo aver considerato aspetti progettuali e di implementazione, è stato necessario integrare uno schema coerente con i requisiti richiesti; pertanto, il **Database** presenta la seguente struttura:



Al suo interno, si trova una tabella denominata "*proposal*", che immagazzina informazioni relative a ciascuna proposta. Questa tabella include un identificatore univoco per la proposta, collegato a un utente tramite l'ID dell'utente e un campo per il nome della proposta. Inoltre, i dettagli completi della proposta sono memorizzati in un campo di tipo BLOB, che memorizzerà direttamente il **contenuto** della proposta formattato con standard HTML.

Accanto a questa, la tabella "*user*" si occupa di archiviare i dati relativi agli utenti. Essa include un identificatore univoco per ogni utente, un indirizzo e-mail e una password **hashata** con il relativo sale, anch'essa memorizzata in formato BLOB. Ulteriori dettagli del profilo, come l'immagine del profilo, sono conservati in un campo MEDIUMBLOB. Inoltre, si utilizza un campo di tipo BLOB per gestire un valore di "salt", fondamentale per **l'archiviazione** sicura delle password.

Infine, la tabella "**cookie**" è dedicata alla gestione dei token cookie. Questa tabella contiene un identificatore per il cookie, un riferimento all'utente attraverso l'ID utente, un token di sessione memorizzato come VARCHAR, e una data di scadenza (*1 giorno*) in formato DATETIME.

## Analisi Statica

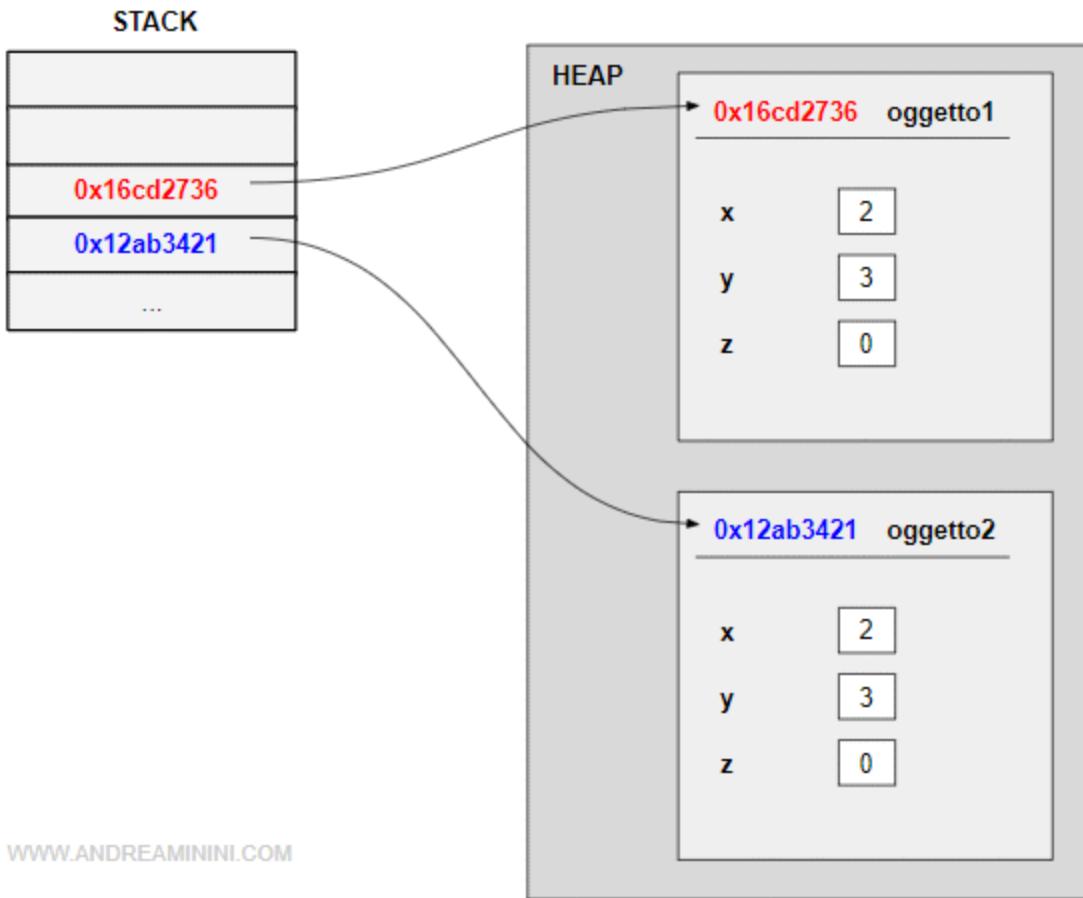
L'analisi statica del codice, nota anche come *Static application Security Testing (SAST)*, è una metodologia di scansione delle vulnerabilità progettata per lavorare sul **codice sorgente** piuttosto che su un **eseguibile** compilato. Gli strumenti di analisi statica del codice ispezionano il codice alla ricerca di indicazioni di **vulnerabilità** comuni, che vengono poi corrette prima che l'applicazione venga rilasciata. Da un lato, offre il vantaggio di identificare problemi di sicurezza, vulnerabilità e difetti strutturali precocemente, migliorando così la qualità del software e riducendo i costi di correzione. D'altra parte, questa metodologia non può rilevare **comportamenti** legati all'esecuzione dinamica, come le prestazioni o le interazioni tra moduli, e spesso genera falsi positivi, richiedendo una revisione manuale per **validare** i risultati. Per questo motivo solitamente viene successivamente integrata un'analisi dinamica che permetterà di testare l'applicazione durante l'esecuzione per ottenere una visione più completa delle problematiche.

Per lo svolgimento corretto di tale analisi, si esamineranno alcune macroaree di interesse che in maniera olistica contribuiscono al raggiungimento di un elevato standard di sicurezza.

## Lifetime dei dati sensibili

Lo **stack** è una struttura di memoria organizzata secondo il principio *Last-In-First-Out (LIFO)*, dedicata principalmente alla gestione delle variabili locali (tra cui generalmente tipi primitivi **int**, **char**, **float**, **double**, **boolean**, **byte**, **short**, e **long**) e alla gestione delle chiamate e dei ritorni delle funzioni. L'allocazione e deallocazione di memoria nello stack avvengono in modo automatizzato e deterministico, caratterizzandosi per rapidità ed efficienza. Lo Stack è assegnato dalla macchina virtuale Java (**JVM**) all'inizio dell'esecuzione del programma.

Al contrario, l'**heap** è una regione di memoria che supporta l'allocazione dinamica e, quindi, la durata degli oggetti allocati può variare. La memoria heap è gestita attraverso un processo di **garbage collection**, il quale è incaricato della liberazione della memoria non più in uso. La durata di vita degli oggetti nella memoria heap non è immediatamente **determinabile**, ma dipende dal comportamento del garbage collector. L'heap è l'area di memoria dove sono memorizzati gli oggetti e il loro stato.



*Quando creo un nuovo oggetto, l'identificatore ossia il nome della variabile (alias) viene aggiunto nello stack insieme all'indirizzo di memoria dell'oggetto.*

Un aspetto cruciale riguarda la gestione degli oggetti di tipo **stringa**. Questi ultimi sono allocati nella memoria **heap** e rimangono in essa fino a quando non viene eseguita una procedura di garbage collection che li rimuove. Tale comportamento implica che, rispetto ad altre variabili, gli oggetti di tipo **stringa** possono rimanere in memoria per un periodo indefinito, aumentando così il rischio di esposizione involontaria di dati sensibili.

A fronte di tali aspetti, è stato necessario implementare alcuni soluzioni di sicurezza.

In questo caso, sia durante la fase di **registrazione** che **login**, la password proveniente dalla **view** di riferimento è stata archiviata in **un'array di byte**, per consentire un'adeguata eliminazione nelle fasi successive.

```
byte[] password = request.getParameter("password").getBytes();
```

Il passo successivo ha riguardato quindi l'eliminazione della password dall'array, utilizzando il metodo **clearBytes** della classe **PasswordManager** durante la fase di reindirizzamento in altre pagine della *Web App*.

```

PasswordManager.clearBytes(password);
PasswordManager.clearBytes(byte_email);
PasswordManager.clearBytes(pad_email);
PasswordManager.clearBytes(byte_encryptedEmail);
email = null;
response.sendRedirect("userNotLoggedLogin.jsp"); // Reindirizzamento in caso di autenticazione fallita
DisplayMessage.showPanel("Password errata! Riprovare");

```

Con il metodo **clearBytes** tutte le posizioni dell'array vengono impostate a “0”

```

public final class PasswordManager {

    //Pulizia dall'array degli elementi
    public static void clearBytes(byte[] password) {
        if (password != null) {

            for (int i = 0; i < password.length; i++) {
                password[i] = 0;
            }
        }
    }
}

```

Un altro aspetto che verrà approfondito successivamente riguarda la **minimizzazione dello scope delle variabili**. È importante notare come viene dichiarata la variabile contatore *i*. Questa è dichiarata all'interno del ciclo for e quindi cesserà di esistere nel momento in cui il ciclo terminerà.

## Cookie

I **cookie** sono dei piccoli file di testo necessari affinché il **server** del sito web che li ha installati sul dispositivo utente, possa ottenere informazioni sulla specifica attività che l'utente compie su quelle pagine web. Ogni volta che quel dispositivo si ricollega al sito **rimanda il cookie al server web**, in maniera tale da riconoscere e tracciare l'attività a distanza di tempo. Possono essere quindi utilizzati per ricordare alcune informazioni che l'utente ha precedentemente inserito nei campi di testo di un sito, come ad esempio *il nome, l'indirizzo, la password, l'e-mail e/o i numeri delle carte di credito*.

L'applicazione richiedeva la progettazione e l'implementazione della funzionalità **Ricordami**: per tale motivo sono stati utilizzati i cosiddetti **Cookie permanenti**, ovvero dei cookie che vengono salvati per un periodo di tempo limitato all'interno del dispositivo visitante e in questo caso, per verificare la validità è stato necessario sincronizzarli anche all'interno del Database.

Questo periodo di tempo in fase di progettazione è stato fissato ad **1 giorno**, in maniera tale da prevenire le diverse tipologie di attacco. Le vulnerabilità di sicurezza possono permettere agli hacker di leggere i dati del *cookie*, che potrebbe essere usato per ottenere l'accesso ai dati degli utenti, o per ottenere l'accesso (con le credenziali dell'utente) al sito web a cui il *cookie* appartiene attraverso **il cross-site scripting (XSS)** e **cross-site request forgery (CSRF)**.

- **XSS:** Un XSS permette a un cracker di inserire o eseguire **codice** lato client (solitamente *javascript*) al fine di attuare un insieme variegato di attacchi quali, ad esempio, raccolta, manipolazione e reindirizzamento di informazioni riservate (*cookie*), visualizzazione e modifica di dati presenti sui server, alterazione del comportamento dinamico delle pagine web, ecc. Per questo motivo è importante sanificare tutti gli input all'interno della Web App.
- Il **Cross-site request forgery**, abbreviato **CSRF** o anche **XSRF**, è una vulnerabilità a cui sono esposti i siti web dinamici quando sono progettati per ricevere richieste da un client senza meccanismi per controllare se la richiesta sia stata inviata intenzionalmente oppure no. Diversamente dal cross-site scripting (XSS), che sfrutta la fiducia di un utente in un particolare sito, il CSRF sfrutta la fiducia di un sito nel browser di un utente.

#### *Esempio*

- *Mentre l'utente è collegato al portale, visita anche un altro sito web creato dall'hacker. Qui, l'utente esegue un'azione qualunque, ad esempio premere un pulsante. In seguito a questa azione, l'hacker invia una richiesta HTTP al portale utilizzato dall'utente e, camuffato con l'identità di quest'ultimo, esegue un'azione dannosa sfruttando il fatto che la sessione è ancora attiva. Per poter raggiungere il suo obiettivo, l'hacker deve soltanto conoscere la richiesta HTTP corretta che, tuttavia, è abbastanza semplice da individuare.*

Come si analizzerà durante la fase di gestione della Sessione, per mitigare gli attacchi **CSRF** si utilizzano i cosiddetti **Token CSRF**.

777 sono progettati per prevenire tali attacchi introducendo un elemento unico e imprevedibile in ogni richiesta effettuata dall'utente. Questi token vengono generati dal server e incorporati nei moduli dell'applicazione Web o nelle richieste **AJAX**. Quando un utente invia un modulo o esegue un'azione che attiva una richiesta **AJAX**, il token **CSRF** viene incluso come parametro o intestazione.

Quando il server riceve la richiesta, verifica il token **CSRF** per garantirne l'autenticità. Se il token manca, non è valido o non corrisponde al valore previsto, il server può rifiutare la richiesta, presupponendo che possa trattarsi di un attacco **CSRF**. Convalidando il token CSRF, il server può distinguere tra richieste legittime avviate dall'utente e richieste dannose avviate da un utente malintenzionato, che ovviamente non conosce il token interno alla sessione.

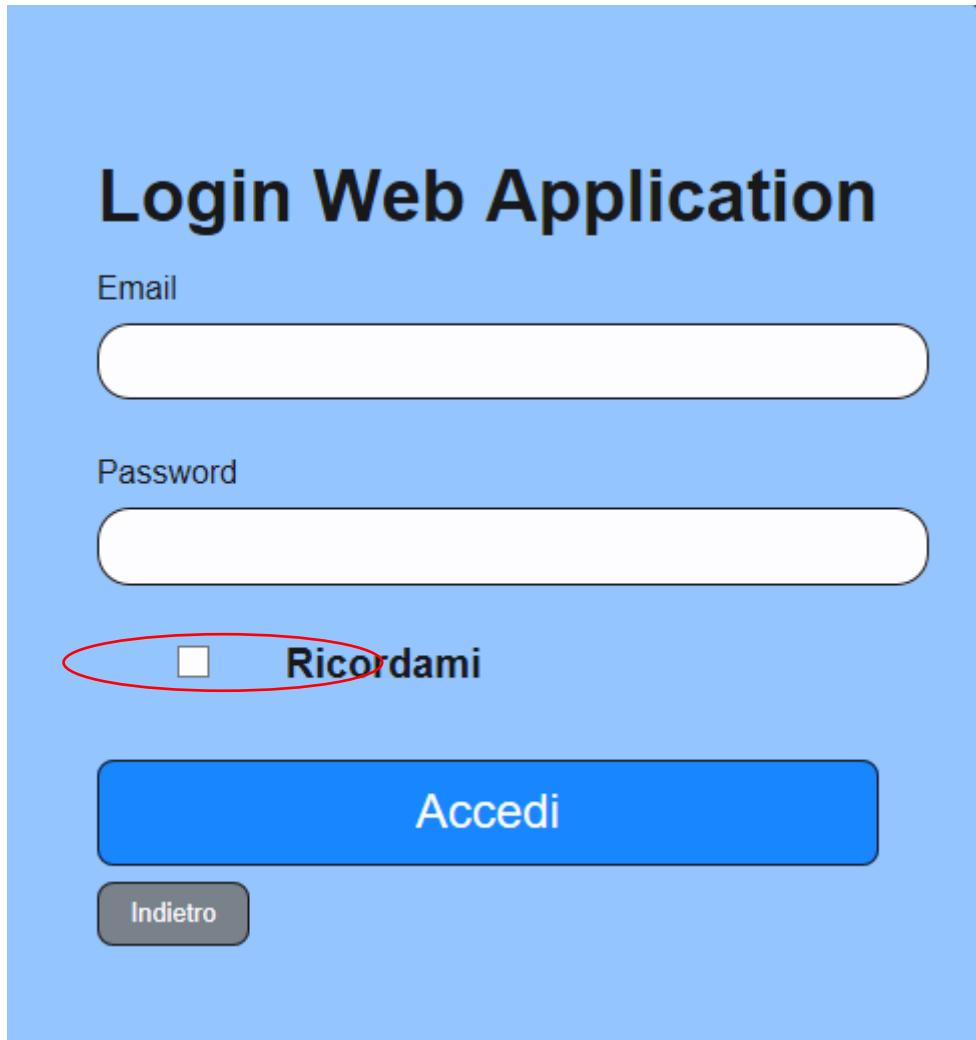
Per quanto riguarda la gestione dei **cookie permanenti**, sono stati testati diversi scenari qui riassunti in una tabella:

ID	Scenario	Descrizione	Azione	Risultato atteso
1	Sessione Attiva con Cookie Valido	L'utente ha una sessione attiva e un cookie "rememberMe" valido.	Naviga verso una pagina protetta.	Accesso consentito senza reindirizzamento.

2	Sessione Attiva con Cookie Scaduto	L'utente ha una sessione attiva, ma il cookie "rememberMe" è scaduto.	La sessione scade.	L'utente viene reindirizzato alla pagina di login.
3	Sessione Attiva senza Cookie	L'utente ha una sessione attiva, ma nessun cookie "rememberMe".	La sessione scade.	L'utente viene reindirizzato alla pagina di login.
4	Sessione Scaduta con Cookie Valido	La sessione è scaduta, ma il cookie "rememberMe" è valido.	Naviga verso una pagina protetta dopo la scadenza della sessione.	La sessione viene ricreata automaticamente usando il cookie e l'utente rimane autenticato.
5	Sessione Scaduta con Cookie Scaduto	La sessione è scaduta e il cookie "rememberMe" è scaduto.	Naviga verso una pagina protetta dopo la scadenza della sessione e del cookie.	L'utente viene reindirizzato alla pagina di login.
6	Sessione Scaduta senza Cookie	La sessione è scaduta e non esiste alcun cookie "rememberMe".	Naviga verso una pagina protetta dopo la scadenza della sessione.	L'utente viene reindirizzato alla pagina di login.
7	Nessuna Sessione con Cookie Valido	Non c'è una sessione attiva, ma il cookie "rememberMe" è valido.	Naviga verso una pagina protetta.	La sessione viene creata e l'utente è autenticato.
8	Nessuna Sessione con Cookie Scaduto	Non c'è una sessione attiva e il cookie "rememberMe" è scaduto.	Naviga verso una pagina protetta.	L'utente viene reindirizzato alla pagina di login.
9	Nessuna Sessione e Nessun Cookie	Non c'è una sessione attiva e non esiste alcun cookie "rememberMe".	Naviga verso una pagina protetta.	L'utente deve effettuare il login.

## Funzionamento meccanismo “Ricordami”

Durante la fase di Login, viene richiesto all'utente se effettuarlo o meno tramite il meccanismo Ricordami.



Successivamente, nella **Servlet** vengono dapprima controllate le credenziali accedendo tramite il **Model** al Database, in seguito se le credenziali risultano valide, viene creata una nuova **Sessione** e un nuovo oggetto della Classe **UserLogged**, indipendentemente dalla scelta dell'utente riguardo la creazione dei Cookie.

Se l'utente seleziona la checkbox **Ricordami** viene invece creato il **cookie di autenticazione**, che avrà come periodo di **validità 1 giorno** e verrà conservato nel **dispositivo** (attraverso il browser) dell'utente, e nel **Database**.

Il cookie viene generato in maniera crittograficamente sicura, per evitare che un attaccante possa **ricrearlo** oppure ottenere informazioni cruciali, attraverso i metodi **generateSecureCookieToken** e **generateFinalToken** della classe **UserLogged**. Il cookie non sarà altro che la concatenazione della **mail cifrata** con il **timestamp** (in riferimento al momento esatto della creazione, a sua volta concatenato con **byte randomici e hashato con algoritmo SHA-256**) successivamente verrà cifrato tramite algoritmo di cifratura simmetrico **AES** e poi verrà calcolato **l'hash** di tale valore. Il metodo **generateFinalToken** invece, prenderà in input tale stringa cifrata e **hashata** e calcolerà nuovamente **l'hash** concatenato con la mail cifrata.

La classe Cookie necessita all'interno del costruttore un valore di tipo **Stringa**, per questo motivo i risultati vengono codificati in **Base64**.

```

private byte[] generateSecureCookieToken() {
    byte[] concatenatedData = new byte[this.byte_encryptedEmail.length + this.timestamp.length];
    System.arraycopy(this.byte_encryptedEmail, 0, concatenatedData, 0, this.byte_encryptedEmail.length);
    System.arraycopy(this.timestamp, 0, concatenatedData, this.byte_encryptedEmail.length, this.timestamp.length);
    byte[] concatenatedDataPadding = Encryption.addPadding(concatenatedData);
    byte[] cookieEncrypted = null;
    try {
        cookieEncrypted = Encryption.encrypt(concatenatedDataPadding);
    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return PasswordManager.concatenateAndHash(cookieEncrypted, this.csrfToken);
}

private byte[] generateFinalToken(byte[] token) {
    return Base64.getEncoder().encode(PasswordManager.concatenateAndHash(this.byte_encryptedEmail, token));
}

```

Il rischio che il cookie invece venga **manipolato** viene mitigato da alcuni parametri di sicurezza qui descritti:

### 1. Utilizzo di HTTPS su Tutte le Connessioni:

- È necessario che tutte le comunicazioni tra il client e il server siano criptate utilizzando HTTPS. Ciò impedirà agli attaccanti di intercettare i cookie durante la trasmissione.

### 2. Impostare il Flag HttpOnly sui Cookie:

- Il flag HttpOnly impedisce ai cookie di essere accessibili tramite JavaScript sul lato client, mitigando il rischio di furto tramite attacchi XSS.

### 3. Utilizzo del flag SetSecure

- Tale flag sui cookie serve a garantire che i suddetti siano trasmessi solo attraverso connessioni sicure, ovvero su protocolli **HTTPS**. Questo significa che un cookie con il flag Secure verrà inviato dal browser al server **solo** se la richiesta viene fatta tramite **HTTPS** e non tramite **HTTP**.

Prima che il cookie venga effettivamente registrato sul **dispositivo** utente, viene correttamente archiviato all'interno del **Database**, in maniera tale da confrontare durante ogni **accesso**, il cookie fornito dal Browser e quello presente nel Database verificandone inoltre, la **validità**. Alla fine del processo l'utente viene reindirizzato alla pagina delle proposte.

```
77     if (LoginDAO.isUserValid(email, password)) {
78
79         UserLogged userlogged = new UserLogged(byte_encryptedEmail);
80
81         HttpSession session = request.getSession(); //Crea una nuova sessione
82         session.setAttribute("email", email);
83         session.setAttribute("login", true);
84         session.setAttribute("csrfToken", Base64.getEncoder().encodeToString(userlogged.getCsrfToken()));
85         session.setAttribute("userLogged", userlogged);
86         session.setMaxInactiveInterval(15*60); // 15 minuti di timeout della sessione
87
88         if (ricordami) {
89
90             byte[] cookie = userlogged.getCookieToken();
91
92             if (StoreCookieDAO.storeCookie(byte_encryptedEmail, cookie)) {
93                 Cookie rememberMeCookie = new Cookie("rememberMe", Base64.getEncoder().encodeToString(cookie));
94                 rememberMeCookie.setMaxAge(COOKIE_MAX_AGE);
95                 rememberMeCookie.setHttpOnly(true);
96                 rememberMeCookie.setSecure(true);
97                 response.addCookie(rememberMeCookie);
98
99
100            // Redirect to logged-in user page
101            response.sendRedirect("userLoggedIndex.jsp");
102            PasswordManager.clearBytes(cookie);
103        } else {
104    
```

Reindirizzamento automatico tramite Cookie permanenti

Per verificare che ogni qualvolta un utente loggato potesse effettivamente accedere alla pagina protetta contenente le proposte o che potesse essere correttamente reindirizzato, è stato necessario l'utilizzo dei **filtri**.

I **filtri** in una **servlet** rappresentano una componente architettonale fondamentale del framework Java per la gestione delle richieste e risposte all'interno di applicazioni web. Essi agiscono come intermediari tra il **client** e il **server**, permettendo l'**intercettazione**, la **modifica** o l'**elaborazione** delle richieste **HTTP** in entrata e delle risposte in uscita. La loro funzione è eminentemente trasversale e modulare, consentendo l'implementazione di logiche aggiuntive che possono essere applicate in modo trasparente e non invasivo rispetto alla logica principale dell'applicazione.

Principalmente è stato utilizzato il metodo **doFilter(ServletRequest request, ServletResponse response, FilterChain chain)** in cui il filtro esegue la sua logica e decide se inoltrare la richiesta alla risorsa successiva (utilizzando **chain.doFilter()**) o modificarla.

```
// Gestione autenticazione tramite cookie se presenti
if (!isLoggedIn) {
    Cookie[] cookies = httpRequest.getCookies();
    if (cookies != null) {
        boolean cookieRemembermeFound=false;
        for (Cookie cookie : cookies) {
            if ("rememberMe".equals(cookie.getName())) {
                cookieRemembermeFound=true;
                byte[] cookieByte = Base64.getDecoder().decode(cookie.getValue());
                System.out.println("Token cookie individuato: " + cookie.getValue());

                //Verifico prima che il cookie esista e che non sia stato eliminato dalla routine
                if(TokenExistsDAO.tokenExists(cookieByte)) {

                    //Verifico che sia valido (data di scadenza)
                    if (IsTokenValidDAO.isTokenValid(cookieByte)) {
                        // Autenticazione tramite cookie riuscita
                        String email = GetEmailFromTokenDAO.getEmailFromToken(cookieByte);

                        byte[] byte_email = ConvertingType.stringToByteArray(email);
                        byte[] pad_email = Encryption.addPadding(byte_email);
                        byte[] byte_encryptedEmail = null;

                        try {
                            byte_encryptedEmail = Encryption.encrypt(pad_email);
                        } catch (Exception e) {
                            e.printStackTrace();
                            DisplayMessage.showPanel("Errore interno, riprovare!");
                            return;
                        }
                    }
                }
            }
        }
    }
}
```

In questo caso, il filtro controlla che la **sessione** sia **valida**, se nel caso così non fosse, ricerca il cookie permanente all'interno del browser. Se individua un cookie valido, verifica la sua presenza all'interno del **Database** e la sua validità. Se queste condizioni risultano valide, il filtro ricreerà una nuova Sessione e reindirizzerà automaticamente l'utente alla pagine delle proposte.

```
//Utente senza cookie token poichè già presente e registrato
UserLogged userlogged = new UserLogged(byte_encryptedEmail,cookieByte);

session = httpRequest.getSession(true); // Crea una nuova sessione
session.setAttribute("email", email);
session.setAttribute("login", true);
session.setAttribute("csrfToken", Base64.getEncoder().encodeToString(userlogged.get

System.out.println("Token individuato la seconda volta: " + Base64.getEncoder().enc
session.setAttribute("userLogged", userlogged);
session.setMaxInactiveInterval(15*60);
PasswordManager.clearBytes(cookieByte);
```

```

// Logica di reindirizzamento basata sull'autenticazione
if (isLoggedin) {
    // Se l'utente è autenticato e tenta di accedere alla pagina di login, reindirizzalo alla pagina principale
    if (isLoginRequest) {
        httpResponse.sendRedirect(INDEX_PAGE);
        return; // Esci dalla funzione dopo il reindirizzamento
    } else {
        // Utente autenticato e non sta tentando di accedere alla pagina di login, continua con la richiesta
        chain.doFilter(request, response);
        return; // Esci dalla funzione dopo aver continuato con la richiesta
    }
} else {
    // Utente non autenticato
    if (isLoginRequest) {
        // Se l'utente non è autenticato ma sta accedendo alla pagina di login, permetti la richiesta
        chain.doFilter(request, response);
    } else {
        // Utente non autenticato e sta tentando di accedere a una pagina protetta, reindirizzalo alla pagina di login
        httpResponse.sendRedirect(LOGIN_PAGE);
    }
}
return; // Esci dalla funzione dopo aver gestito la richiesta
}

```

*Meccanismo per verificare le richieste dell'utente ed effettuare il reindirizzamento corretto.*

Tutti i casi analizzati precedentemente non consideravano però, un aspetto importante: alcuni browser tendono per motivi di sicurezza ad eliminare i cookie dopo la chiusura di quest'ultimo. Per questo motivo è stata implementata un routine con **frequenza** di 24 ore, per evitare che nel Database vengano conservati cookie non più esistenti nel Browser, i cosiddetti **Cookie Orfani**. Questo permette ad ogni avvio del Server Web, di identificare cookie scaduti e rimuoverli automaticamente per non inficiare sulla spazio disponibile all'interno del Database.

```

12 @ThreadSafe
13 @WebListener // Annotazione che registra il listener
14 public final class ApplicationStartupListener implements ServletContextListener {
15
16     // Esecutore per il task di pulizia
17     private ScheduledExecutorService scheduler;
18
19     @Override
20     public void contextInitialized(ServletContextEvent sce) {
21         // Creazione dell'esecutore con un singolo thread
22         scheduler = Executors.newSingleThreadScheduledExecutor();
23
24         // Pianifica la pulizia dei token scaduti ogni 24 ore
25         scheduler.scheduleAtFixedRate(new Runnable() {
26             @Override
27             public void run() {
28                 // Logica di pulizia dei token
29                 CleanExpiredTokenDAO.cleanupExpiredToken();
30             }
31         }, 0, 24, TimeUnit.HOURS);
32     }
33
34     @Override
35     public void contextDestroyed(ServletContextEvent sce) {
36         // Interrompe l'esecutore quando il contesto viene distrutto
37         if (scheduler != null) {
38             scheduler.shutdownNow();
39         }
40     }
41 }

```

## Sessione HTTPS/TLS

A differenza dei cookie, una **sessione** è una connessione lato server che associa una serie di **interazioni** tra il client (spesso un browser web) e il server a **un'identità univoca temporanea**. Questa identità è rappresentata da un identificatore di sessione, generalmente un token univoco e casuale, generato dal server alla creazione della sessione. Le informazioni di sessione vengono archiviate sul server e vengono recuperate a ogni richiesta successiva del client tramite l'identificatore, che viene inviato dal client come parte della richiesta. Il vantaggio principale della sessione è che le informazioni rimangono sicure, poiché risiedono sul server e non sono visibili o accessibili direttamente al client. Tuttavia, le sessioni sono temporanee e spesso limitate a una durata specifica o a una determinata inattività da parte dell'utente.

**Una nuova sessione viene creata in due casi:**

1. *L'utente effettua la prima volta il login con le credenziali corrette*
2. *L'utente visita la pagina di login senza effettuarlo con un cookie ancora valido*

Per evitare attacchi di tipo **CSRF** è stato necessario impostare un time-out di sessione pari a **15** minuti con in aggiunta un token **CSRF**.

```
HttpSession session = request.getSession(); //Crea una nuova sessione
session.setAttribute("email", email);
session.setAttribute("login", true);
session.setAttribute("csrfToken", Base64.getEncoder().encodeToString(userlogged.getCsrfToken()));
session.setAttribute("userLogged", userlogged);
session.setMaxInactiveInterval(15*60); // 15 minuti di timeout della sessione
```

Difatti, il metodo **session.setMaxInactiveInterval(int seconds)** in una servlet imposta il tempo massimo di **inattività**, in secondi, dopo il quale una sessione diventa inattiva e viene invalidata automaticamente dal server.

Quando un client non invia richieste al server per un intervallo di tempo pari o superiore al valore specificato, il server invalida la sessione associata. Questo significa che tutte le informazioni associate alla sessione vengono perse, e una nuova sessione dovrà essere creata se il client invia una nuova **richiesta**. L'impostazione del tempo massimo di inattività è importante per motivi di sicurezza, poiché limita la durata di una sessione non utilizzata, riducendo il rischio che sessioni dimenticate o non chiuse vengano utilizzate in modo non autorizzato.

Un controllo più diretto è stato anche effettuato lato client attraverso **Javascript**, utilizzando nello specifico la libreria **Jquery**:

```
var inattivitàTimer; // Variabile globale

function resetInattivitàTimer() {
    if (inattivitàTimer) {
        clearTimeout(inattivitàTimer); // Pulisce il timer precedente se esiste
    }

    inattivitàTimer = setTimeout(function() {
        $.ajax({
            type: 'POST',
            url: 'LogoutServlet',
            success: function() {
                window.location.href = "userNotLoggedIndex.jsp";
            }
        });
    }, 900000); // Timeout di 15 minuti (uguale per la sessione)
}

// Eventi che indicano attività dell'utente
$(document).on('mousemove keypress click', function() {
    resetInattivitàTimer();
});

// Imposta inizialmente il timer di inattività quando la pagina si carica
resetInattivitàTimer();

});
```

In questo caso, il timer di **sessione** e il timer impostato da **Javascript** vengono avviati in maniera pseudo-sincronizzata, in maniera tale che una volta scaduta la sessione, venga effettuato un logout automatico.

Il Token **CSRF** così come il token cookie, viene generato in maniera **crittograficamente sicura** attraverso la classe **UserLogged** e il metodo *generateCSRFToken()*.

```
private byte[] generateCsrfToken() {

    final int CSRF_TOKEN_LENGTH = 32; // 32 bytes * 8 = 256 bits
    final SecureRandom secureRandom = new SecureRandom();
    byte[] tokenBytes = new byte[CSRF_TOKEN_LENGTH];
    secureRandom.nextBytes(tokenBytes);
    byte[] tokenBytesPadding = Encryption.addPadding(tokenBytes);
    try{
        PasswordManager.clearBytes(tokenBytes);
        return Encryption.encrypt(tokenBytesPadding);
    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    return null;
}
```

La particolarità di tale **Token** è che non rimarrà invariato durante tutta la sessione, ma bensì ogni qualvolta l'utente caricherà una nuova proposta, verrà generato un nuovo token che verrà trasmesso tramite richiesta **AJAX** alla **View** e contemporaneamente verrà settato come nuovo parametro di **sessione**, per evitare che venga utilizzato lo stesso durante tutti i caricamenti.

```
// Controlla solo le richieste POST, PUT e DELETE per CSRF
if ("POST".equalsIgnoreCase(method) || "PUT".equalsIgnoreCase(method) || "DELETE".equalsIgnoreCase(method)) {
    System.out.println("Dentro il filtro POST 2");
    char[] csrfToken = ConvertingType.parseStringToArray(request.getParameter("csrfToken")); // Prende il token dal file di richiesta json
    char[] sessionCsrfToken = (session != null) ? ConvertingType.parseStringToArray((String)session.getAttribute("csrfToken")) : null;

    if (csrfToken == null || sessionCsrfToken == null || !ConvertingType.areCharArraysEqual(sessionCsrfToken, csrfToken)) {
        boolSecureCsfr=false;
    }else {
        boolSecureCsfr=true;
    }
}
```

In questa prima fase, viene controllata che la richiesta sia POST e dopodiché viene effettuato un controllo sul token **CSRF**, in maniera tale da evitare attacchi CSRF provenienti da sessioni non autentiche.

Successivamente, una volta che la **proposta** è stata correttamente caricata all'interno del **Database**, verrà settato un nuovo **token** sia nella sessione corrente, che nella **view** di riferimento (*accessibile soltanto agli utenti loggati*).

```
userlogged.setCsrfToken();
byte[] newCsrfToken=userlogged.getCsrfToken();
session.setAttribute("csrfToken", Base64.getEncoder().encodeToString(newCsrfToken));

System.out.println("New CSRF TOKEN " + Base64.getEncoder().encodeToString(newCsrfToken));

response.setHeader("X-CSRF-Token", Base64.getEncoder().encodeToString(newCsrfToken));
// Invia il contenuto filtrato come risposta AJAX
response.setContentType("text/plain");

response.getWriter().write(cleanedHtml);
PasswordManager.clearBytes(newCsrfToken);
DisplayMessage.showPanel("La proposta è stata correttamente caricata!");
```

La pagina contenente le proposte, la stessa che permette anche il caricamento, è ovviamente una pagina protetta, accessibile soltanto agli **utenti autorizzati**. Per questo motivo è stato necessario anche in questo caso utilizzare un filtro, che controllasse che qualsiasi accesso fosse contingentato da opportuni controlli:

```
@Override  
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  
throws IOException, ServletException {  
  
    final String LOGIN_PAGE = "userNotLoggedLogin.jsp";  
  
    HttpServletRequest httpRequest = (HttpServletRequest) request;  
    HttpServletResponse httpResponse = (HttpServletResponse) response;  
  
    // Imposta le intestazioni di cache per impedire il caching delle pagine protette  
    httpResponse.setHeader("Cache-Control", "no-cache, no-store, must-revalidate");  
    httpResponse.setHeader("Pragma", "no-cache");  
    httpResponse.setDateHeader("Expires", 0);  
  
    // Recupera la sessione corrente, se esiste  
    HttpSession session = httpRequest.getSession(false);  
    boolean isLoggedInProtected = (session != null && session.getAttribute("email") != null);  
  
    // Stampa di debug per la sessione e l'email  
    if (session != null) {  
        System.out.println("isLoggedInProtected: " + isLoggedInProtected);  
        System.out.println("EmailProtected: " + session.getAttribute("email"));  
    }  
  
    // Se l'utente non è loggato, reindirizza alla pagina di login  
    if (!isLoggedInProtected) {  
        httpResponse.sendRedirect(LOGIN_PAGE);  
        return;  
    }  
  
    chain.doFilter(request, response);  
}
```

Come si può notare, viene rilevata la sessione corrente e viene verificata se è presente la mail configurata durante la fase di **Login**: se la sessione è **valida**, l'utente viene reindirizzato alla pagina protetta, altrimenti alla pagina di login.

Inoltre, i tag iniziali vengono utilizzati per Impedire il caching di pagine con informazioni sensibili riduce il rischio che informazioni private siano accessibili da un'altra sessione dell'utente o che siano visibili su dispositivi pubblici o condivisi.

Durante la fase di **Logout**, la sessione viene invalidata in maniera sicura e i cookie cancellati. Se il logout avviene in maniera automatica – *come visto precedentemente* - in questo caso i cookie vengono mantenuti fino al giorno di scadenza **predeterminato** in fase di **Login**.

```

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    // TODO Auto-generated method stub

    HttpSession session = request.getSession(false);
    if (session != null) {
        session.invalidate();
        DisplayMessage.showPanel("Sessione invalidata correttamente!");
    }

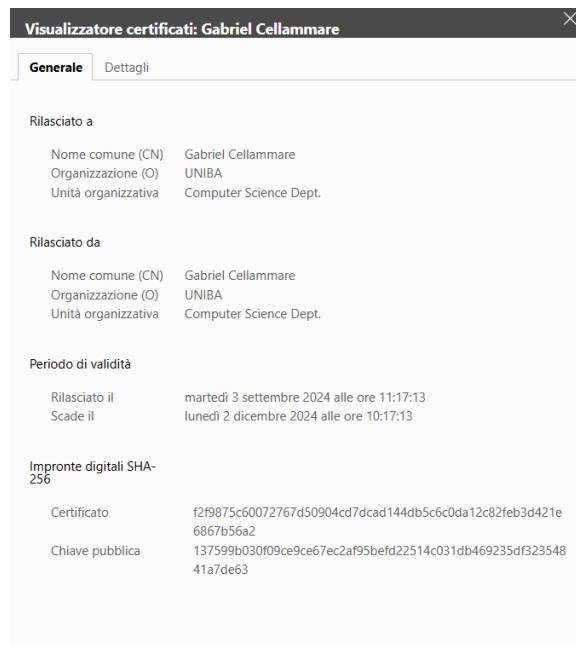
    Cookie[] cookies = request.getCookies();
    if (cookies != null) {
        for (Cookie cookie : cookies) {
            if ("rememberMe".equals(cookie.getName())) {
                byte[] cookieByte = Base64.getDecoder().decode(cookie.getValue());
                if (DeleteTokenDAO.deleteToken(cookieByte)) {
                    // Rimuove il cookie dal browser
                    cookie.setMaxAge(0);
                    cookie.setHttpOnly(true);
                    cookie.setSecure(true);
                    response.addCookie(cookie);
                }
                DisplayMessage.showPanel("Logout manuale effettuato correttamente!");
            }
            PasswordManager.clearBytes(cookieByte);
        }
    } else {
        DisplayMessage.showPanel("Logout manuale senza cookie effettuato correttamente!");
    }
}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    // TODO Auto-generated method stub

    // Invalido solo la sessione, non distruggo i cookie
    HttpSession session = request.getSession(false);
    if (session != null) {
        session.invalidate();
        DisplayMessage.showPanel("Sessione invalidata correttamente!");
    }
    DisplayMessage.showPanel("Logout automatico effettuato correttamente!");
}

```

È inoltre importante far sì che la comunicazione sia cifrata attraverso **HTTPS**. Grazie a Java Key Tool, è possibile generare un certificato auto firmato poiché permette la gestione dei certificati digitali, garantendo che le applicazioni Java possano partecipare a connessioni sicure **HTTPS** tramite il protocollo **TLS/SSL**, assicurando una protezione adeguata durante la comunicazione. È usato principalmente per gestire i **keystore**, archivi sicuri dove vengono conservate le coppie di chiavi, i certificati pubblici e le chiavi private. Questi certificati sono cruciali per abilitare la **comunicazione sicura** attraverso il protocollo **TLS/SSL** (Transport Layer Security/Secure Sockets Layer), garantendo **autenticazione, riservatezza e integrità dei dati**.



## JDBC

**JDBC** (*Java Database Connectivity*) è un'**API** (Application Programming Interface) di Java che consente la comunicazione tra un'applicazione Java e un database relazionale. Introdotta come parte del JDK, JDBC rappresenta un'interfaccia standard per eseguire operazioni come la **connessione** a un database, l'invio di **query SQL** e la gestione dei **risultati** ottenuti.

Il suo ruolo è fondamentale nell'integrazione tra il mondo Java e i database, poiché offre un'astrazione omogenea che permette alle applicazioni di interagire con database di diversi tipi senza dover riscrivere il codice per ogni implementazione specifica.

## Prepared Statements (PS) e la loro importanza

Le **Prepared Statements** (PS) sono una componente chiave di JDBC e rivestono una grande importanza, soprattutto dal punto di vista della sicurezza e delle prestazioni. A differenza delle semplici **query SQL concatenate come stringhe**, le **Prepared Statements** sono precompilate dal database e accettano parametri che possono essere inseriti **dinamicamente**. Questo meccanismo presenta numerosi vantaggi.

### 1. Protezione contro gli attacchi SQL Injection:

Gli attacchi di tipo SQL Injection rappresentano una delle minacce più comuni quando un'applicazione accetta input dall'utente e lo utilizza direttamente nelle query SQL. Le **Prepared Statements** mitigano questo rischio separando la logica della query dai dati forniti. I parametri inseriti tramite una Prepared Statement sono trattati come dati puri e non come parte della query SQL stessa, rendendo impossibile per un utente malevolo manipolare la query.

```
db.query_insertCookie=INSERT INTO cookie (id_user, token, expiry_date) VALUES (?, ?, NOW() + INTERVAL 1 DAY);
```

*Esempio di una Prepared Statements*

Ad esempio, se un'applicazione accetta un nome utente come input e lo inserisce in una query SQL normale, un utente potrebbe inserire codice SQL dannoso per interferire con l'esecuzione della query. Utilizzando una **Prepared Statement**, il nome utente verrebbe inserito come parametro sicuro, prevenendo qualsiasi tentativo di SQL Injection.

### 2. Efficienza e prestazioni migliorate:

Le **Prepared Statements** possono essere riutilizzate più volte con parametri diversi senza la necessità di ricompilare la query SQL ogni volta. Questo riduce il carico computazionale sul database, migliorando le prestazioni, specialmente nelle applicazioni che eseguono molte query simili. Il database può memorizzare la query precompilata in una cache, eseguendola più rapidamente per ogni nuovo set di parametri.

### 3. Maggiore leggibilità e manutenzione del codice:

Utilizzando le **Prepared Statements**, il codice risulta più leggibile e manutenibile. Anziché concatenare stringhe di SQL, spesso lunghe e difficili da gestire, le **PS** offrono una struttura chiara e definita in cui la query viene separata dai parametri. Questo non solo facilita la lettura, ma rende anche più semplice modificare la logica delle query o gestire i dati forniti dall'utente. In questo caso, i parametri verranno preparati ad Hoc utilizzando i tipi primitivi (e non solo)

di Java, in maniera tale che il JDBC conosca la tipologia di variabile accettata, riducendo drasticamente i rischi di SQL Injection.

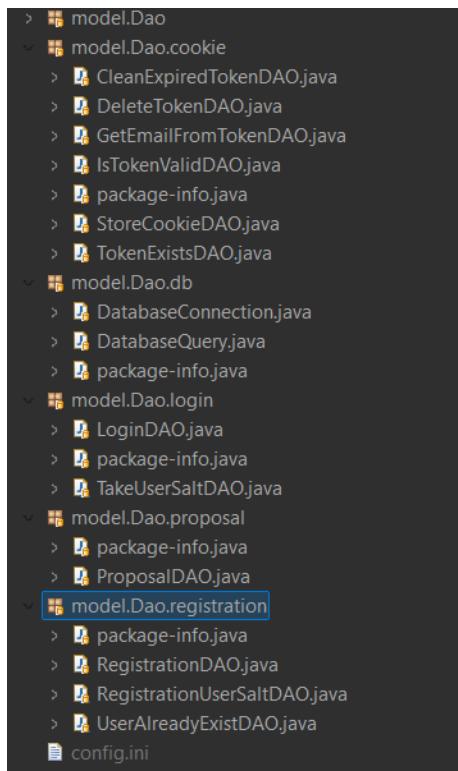
Per evitare ulteriormente iniezione di codice **malevolo** e come visto in precedenza attacchi **XSS**, sono stati sanificati tutti i campi di **input**; questo, è stato possibile grazie alla classe **EmailChecker** che attraverso un'espressione regolare gestisce quelli che è il campo **E-mail** presente sia durante la fase di Login che di registrazione. Ulteriori **input malevoli** potrebbero provenire dal **file HTML** contenente la proposta che difatti viene anche renderizzato dalla **Web App**: vedremo nella sezione apposita quali sono stati gli strumenti adottati per evitare ciò.

```
package application.util.fileChecker;  
  
import java.util.regex.Pattern;  
  
@Immutable  
public final class EmailChecker {  
  
    public final static boolean isValidEmail(String email) {  
  
        final Pattern ptr = Pattern.compile("^(?=.{1,64}@)[A-Za-z0-9_-]+(\\.[A-Za-z0-9_-]+)*@"  
            + "[^-][A-Za-z0-9-]+(\\.[A-Za-z0-9-]+)*(\\.[A-Za-z]{2,})$");  
        return ptr.matcher(email).matches();  
    }  
}
```

Inoltre, ogni **DAO**<sup>2</sup> è stato separato per ogni query o gruppo di operazioni correlate perché questo approccio migliora la **modularità** e la **manutenibilità** del codice. Separando la logica di accesso ai dati in DAO **distinti**, si garantisce una maggiore chiarezza e organizzazione del codice, facilitando la gestione delle modifiche e il riutilizzo. Ogni DAO può essere focalizzato su una specifica entità o insieme di operazioni, rendendo l'applicazione più comprensibile e più semplice da testare o estendere in futuro.

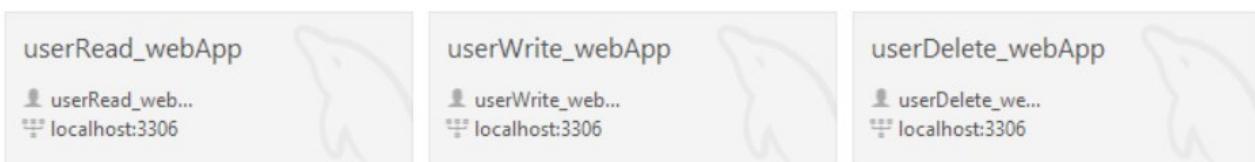
---

<sup>2</sup> Il pattern **DAO** (Data Access Object) è un pattern architettonale che viene spesso utilizzato nelle applicazioni Java per separare la logica di accesso ai dati dal resto dell'applicazione.



Come si può notare, le query SQL sono memorizzate in un file di configurazione chiamato **config.ini** poiché piuttosto che essere codificate direttamente nel codice sorgente (*hard coded*), si introducono una serie di benefici significativi semplificando ulteriormente la **gestione** dell'accesso al database e permettendo di cambiare facilmente i dettagli di **connessione** in base all'ambiente in cui l'applicazione viene eseguita.

La misura di sicurezza che mitiga ulteriormente quelli che potrebbero essere attacchi diretti al **Database**, riguarda l'utilizzo di connessioni separate per operazioni di **lettura, scrittura e cancellazione**. Questo garantisce un'ulteriore divisione delle **responsabilità** e della sicurezza a livello di database, dal momento che ogni operazione può essere eseguita con i minimi privilegi necessari, riducendo il rischio di errori o vulnerabilità.



Dal punto di vista della sicurezza, **connessioni con privilegi distinti** limitano l'accesso alle operazioni **critiche**, riducendo il rischio di abuso o errore. A livello di prestazioni, separare le connessioni può ottimizzare il carico del database, permettendo di gestire in modo più efficiente operazioni ad alto volume o che richiedono diverse priorità. Questo approccio garantisce anche una maggiore **robustezza**, consentendo di applicare politiche di timeout, pooling o transazioni personalizzate per ciascun tipo di operazione.

```

public final class LoginDAO {
    public static boolean isValid(String email, byte[] password) {
        boolean status = false;
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            try( Connection con_read = DatabaseConnection.getConnectionRead()){
                int id_user=TakeUserIdDAO.takeUserId(con_read, email);
                if(id_user>0) {
                    Blob userSaltBlob = TakeUserSaltDAO.takeUserSalt(con_read, id_user);
                    if (userSaltBlob != null) {
                        byte[] sale = userSaltBlob.getBytes(1, (int) userSaltBlob.length());
                        byte[] newPassword = PasswordManager.concatenateAndHash(password, sale);
                        try(PreparedStatement psUser = con_read.prepareStatement(DatabaseQuery.getSelectUserQuery())){
                            psUser.setString(1, email);
                            psUser.setBytes(2, newPassword);
                            try(ResultSet rsUser = psUser.executeQuery()){
                                status = rsUser.next();
                                if (status) {
                                    System.out.println("Utente trovato");
                                } else {
                                    DisplayMessage.showPanel("Errore nell'inserimento dei dati dell'utente. Riprova"); //VALUTARE DI ELIMINARE?
                                }
                            }
                        } else {
                            System.out.println("Salt è nullo");
                        }
                    } else {
                        DisplayMessage.showPanel("ID Non valido. Riprovare con un mail corretta!");
                        System.out.println("Nessun risultato trovato per l'utente: " + email);
                    }
                }
            } catch(ClassNotFoundException|SQLException e) {
                DisplayMessage.showPanel("Si è verificato un problema di connessione!");
                e.printStackTrace();
            }
        }
        return status;
    }
}

```

Esempio del DAO riguardante la fase di Login

## Crittografia

La **crittografia** è fondamentale per garantire la **riservatezza** e l'**integrità** delle **informazioni** trasmesse o memorizzate in una **web application**, inclusi **cookie** e dati **sensibili**. Cifrando queste informazioni, si previene l'accesso non autorizzato e si protegge dai tentativi di attacco, come il furto di sessioni o la manomissione dei dati. Gli standard **NIST** (*National Institute of Standards and Technology*) raccomandano algoritmi robusti come **AES** (*Advanced Encryption Standard*), che garantisce elevata sicurezza.

In particolare, l'algoritmo **AES** in modalità **CBC** (*Cipher Block Chaining*) è comunemente usato per crittografare blocchi di dati in modo sicuro. CBC combina ciascun blocco con il blocco precedente prima di essere cifrato, aumentando la sicurezza complessiva. Tuttavia, poiché AES opera su blocchi fissi (128 bit), il **padding** (ad esempio, lo schema **PKCS#7**) è necessario per riempire l'ultimo blocco se i dati non sono multipli della lunghezza richiesta. Questo garantisce che la cifratura sia sempre eseguita su blocchi di lunghezza **uniforme**, prevenendo vulnerabilità legata alla lunghezza variabile dei dati.

L'uso della crittografia secondo gli standard NIST è cruciale per evitare attacchi comuni, come l'analisi dei pattern nei dati o nei cookie non cifrati, e garantire la sicurezza delle comunicazioni e delle operazioni critiche nella web application.

La classe **Encryption** è responsabile della cifratura e decifratura delle informazioni dell'intera **Web App**:

```

public final class Encryption {

    //Cifratura utilizzando AES in modalità CBC
    public static byte[] encrypt(byte[] data) throws Exception {

        char[] AES_KEY=Encryption.readAesKey();
        SecretKey key = Encryption.getSecretKey(AES_KEY);

        if (key != null) {

            char [] AES_IV = Encryption.readAES_IV();
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
            cipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(ConvertingType.charToBytes(AES_IV)));
            Arrays.fill(AES_KEY, '\0');
            Arrays.fill(AES_IV, '\0');
            return cipher.doFinal(data);
        } else {
            Arrays.fill(AES_KEY, '\0');
            System.out.println("Chiave non valida.");
            return null;
        }
    }
}

```

Il processo di cifratura utilizza l'algoritmo **AES** (*Advanced Encryption Standard*) in modalità CBC (*Cipher Block Chaining*) per garantire la sicurezza dei dati. La cifratura si svolge in più fasi, a partire dalla lettura della chiave **AES** e del vettore di inizializzazione (**IV**) da un file di configurazione. La chiave, rappresentata da un **array di caratteri**, viene decodificata e convertita in un oggetto **SecretKey**.

L'algoritmo AES viene inizializzato in modalità CBC, che cifra ogni blocco di dati insieme al blocco precedente, legando così i blocchi cifrati e rendendo più complesso per un eventuale attaccante dedurre informazioni dai singoli blocchi. L'inizializzazione del cifrario in modalità **ENCRYPT\_MODE** viene completata con la chiave e il vettore di inizializzazione, che aggiunge ulteriore casualità al processo di cifratura, rendendo ogni output cifrato unico, anche se i dati di input sono uguali.

Il padding di tipo **PKCS5Padding** viene utilizzato per gestire dati di lunghezza non divisibile esattamente per la dimensione del blocco AES (128 bit), garantendo che i dati vengano sempre cifrati in blocchi completi. Una volta cifrati i dati, la funzione restituisce l'output cifrato, che può essere memorizzato o trasmesso in modo sicuro in un array di byte.

Il codice implementa pratiche di sicurezza come l'azzeramento della memoria in cui vengono memorizzate le chiavi dopo l'uso, riducendo il rischio che informazioni sensibili restino disponibili in memoria per eventuali attacchi.

***Il processo è quindi finalizzato a garantire la riservatezza dei dati, proteggendoli da accessi non autorizzati e rispettando standard crittografici moderni.***

Il processo di decifratura utilizza il medesimo meccanismo, in modalità decifratura.

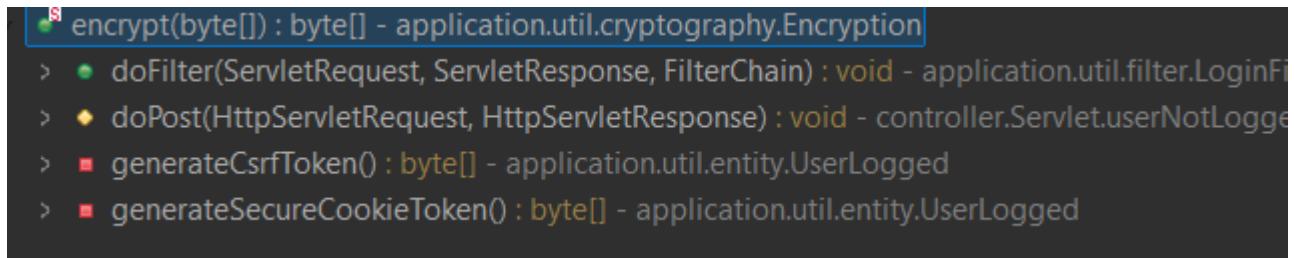
I metodi necessari, dichiarati privati alla classe, sono tre:

1. **private static char[] readAesKey()**: Questa funzione legge la chiave AES dal file di configurazione (*config.ini*). Il metodo utilizza un oggetto **Properties** per caricare le proprietà del file e cerca una proprietà specifica associata alla chiave AES (aes.key). La chiave, memorizzata in formato stringa, viene convertita in un **array di caratteri** attraverso il metodo ausiliario *ConvertingType.parseStringtoCharArray()*. Questo è un approccio utile per trattare le chiavi in modo sicuro, poiché un array di caratteri **può essere azzerato dopo l'uso**,

diversamente dalle stringhe che restano in memoria fino alla garbage collection.

2. **private static char[] readAES\_IV()**: Simile a `readAesKey()`, questa funzione legge il vettore di inizializzazione (*IV*) dal file di configurazione, che è necessario per l'algoritmo **AES** in modalità CBC. L'*IV* è un altro elemento critico della cifratura in CBC, poiché aggiunge casualità al processo, garantendo che due dati uguali cifrati con la stessa chiave producano risultati differenti se inizializzati con IV diversi. Anche qui, l'*IV* viene trattato come un array di caratteri per poter essere cancellato dalla memoria in modo sicuro.
3. **private static SecretKey getSecretKey(char[] AES\_KEY)**: Questa funzione **converte la chiave AES**, memorizzata come un array di caratteri, in un oggetto **SecretKey**, utilizzando la classe `SecretKeySpec`. La chiave in formato stringa viene prima decodificata da **base64**, un formato comunemente utilizzato per rappresentare i byte di una chiave in un modo leggibile e facilmente trasmissibile. L'oggetto `SecretKeySpec` viene poi utilizzato per definire la chiave simmetrica con l'algoritmo AES. Questo è un passo cruciale poiché la chiave in **forma binaria** è ciò che consente all'algoritmo di funzionare correttamente, e la sua rappresentazione tramite `SecretKey` è necessaria per poterla utilizzare nelle operazioni di cifratura e decifratura.

La cifratura dei dati viene utilizzata quindi in diversi casi:



- *L'email viene cifrata per aggiungere entropia al processo di costruzione dei cookie*
- *Il token CSRF viene costruito a partire da un numero generato in maniera crittograficamente sicura e successivamente cifrato*
- *Il token Cookie viene generato a partire da elementi intrinseci dell'utente, successivamente cifrati e hashati.*

Ricordiamo come l'adozione degli standard NIST (*National Institute of Standards and Technology*) è di **fondamentale** importanza nel contesto della sicurezza informatica, in particolare per quanto riguarda la cifratura e la gestione sicura dei dati. Il NIST stabilisce linee guida rigorose e ben documentate per la creazione, l'implementazione e la valutazione di algoritmi crittografici, rendendoli affidabili e ampiamente accettati in ambito internazionale.

## Gestione File

Il processo di caricamento di una **proposta progettuale** o dell'immagine del profilo da parte dell'utente avviene attraverso una procedura **strutturata e rigorosa** che coinvolge un duplice controllo sui file inviati. Tali controlli, come stabilito nel codice sottostante, sono finalizzati a garantire **l'integrità e la sicurezza del sistema**, assicurandosi che i file caricati rispettino precisi criteri tecnici e che non ci siano tentativi di manipolazione malevola.

## Immagine del profilo

Quando un utente carica un'immagine del profilo, il sistema verifica innanzitutto se il file è effettivamente un'immagine **valida**. Questo avviene attraverso un controllo dell'estensione del file, che deve essere limitata ai formati comuni per le immagini, come "jpeg", "jpg" o "png". Tuttavia, poiché l'estensione di un file può essere facilmente modificata in maniera fraudolenta, il sistema si avvale di **Apache Tika**, una **libreria** di analisi del contenuto che consente di determinare il **content-type** reale del file. Attraverso questo processo, si verifica che il file sia effettivamente un'immagine, indipendentemente dall'estensione che potrebbe essere stata **modificata**. Inoltre, viene imposto un limite sulla dimensione dell'immagine, fissato a **cinque MB**, al fine di evitare caricamenti eccessivi che potrebbero compromettere le prestazioni del sistema e non essere **adeguati** alle possibilità di **archiviazione** del Database, dipendente in casi più reali, **dall'infrastruttura** sottostante.

```
public static boolean checkImageFile(Part filePart, byte[] checksumOriginal) throws IOException{
    final long maxSizeInBytes = 5 * 1024 * 1024; // 5 MB
    Tika tika = new Tika();
    boolean contentTypeBool=false;
    boolean check=false;

    byte[] lastChecksum = Encryption.calculateChecksumFromPart(filePart);

    check=Arrays.equals(checksumOriginal, lastChecksum);

    if(!check) {
        DisplayMessage.showPanel("Non è stato possibile terminare la registrazione, le immagini profilo non risultano uguali!");
        return false;
    }

    if (filePart != null && filePart.getSize() > 0) {

        try {
            ImageProfileFileChecker.printMetadata(filePart, tika);
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        String fileName = Paths.get(filePart.getSubmittedFileName()).getFileName().toString();

        String fileExtension = fileName.substring(fileName.lastIndexOf(".") + 1).toLowerCase();

        if ("jpeg".equals(fileExtension) || "jpg".equals(fileExtension) || "png".equals(fileExtension)) {

            if (filePart.getSize() > maxSizeInBytes) {
                DisplayMessage.showPanel("L'immagine selezionata supera la dimensione massima consentita che è di 5 MB");
                return false;
            }

            try(InputStream inputstream = filePart.getInputStream()){
                String contentType = tika.detect(inputstream);

                if (contentType != null && contentType.startsWith("image/")) {
                    contentTypeBool=true;
                } else {
                    contentTypeBool=false;
                    DisplayMessage.showPanel("Il file non è un'immagine valida.");
                }
            }catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Questa classe si occupa di verificare l'immagine del profilo caricata dall'utente, ovvero:

- **Verifica del checksum:** Confronta il checksum originale del file con quello dell'immagine caricata utilizzando `Encryption.calculateChecksumFromPart`. Se i due checksum non corrispondono, il **caricamento fallisce**. Questo passaggio serve a evitare manomissioni (*vedremo il motivo legato a TOCTOU più avanti*).

- **Controllo delle dimensioni e dell'estensione:** Verifica che il file caricato sia un'immagine con estensione valida ("jpeg", "jpg", "png") e che la dimensione non superi i 5 MB.
- **Verifica del *content-type*:** Con l'uso di *Apache Tika*, il metodo verifica che il *content-type* dell'immagine corrisponda effettivamente a **un'immagine** (e non a un altro tipo di file **camuffato** come immagine). Questo serve a prevenire attacchi basati su file con estensioni false.
- **Stampa dei metadati:** Il metodo *printMetadata* viene chiamato per ottenere e visualizzare i metadati dell'immagine caricata. Questo include informazioni come il **tipo** di file e altri dettagli estratti utilizzando *Apache Tika* per ottenere e stampare i metadati dell'immagine caricata. È utile per estrarre informazioni aggiuntive sul file, come il tipo di contenuto, la data di creazione, e altre **proprietà** rilevanti.

Il checksum in questo caso viene calcolato in due istanti diversi, ovvero:

- I. *Durante l'acquisizione dal form di registrazione*
- II. *Prima del caricamento dell'immagine nel Database*

All'interno di questa finestra temporale - **ovvero durante l'apertura del file e la conseguenziale determinazione della tipologia del file stesso** - il checksum viene controllato, per verificare che si tratti effettivamente del file di riferimento.

Il calcolo del checksum serve a prevenire attacchi **TOCTOU**. Il checksum rappresenta un'impronta digitale **unica** del contenuto del file. Se un file viene **modificato** dopo il calcolo del checksum, il checksum risultante sarà **diverso**, rivelando che il file è stato **alterato**. Questo meccanismo assicura che i file caricati e processati siano esattamente quelli che sono stati originariamente verificati.

Il **TOCTOU** (*Time Of Check to Time Of Use*) è un tipo di vulnerabilità che si verifica quando c'è una finestra di tempo tra il momento in cui **un file o risorsa viene controllato** (ad esempio per la validità o l'integrità) e **il momento in cui viene effettivamente utilizzato**. Durante questo intervallo, un attore malevolo potrebbe **modificare** il file, causando l'uso di dati che non sono stati verificati.

#### Esempio di TOCTOU:

1. *Un utente carica un file, e il sistema ne verifica l'estensione e il checksum.*
2. *Prima che il file venga effettivamente utilizzato, l'attaccante modifica il file o sostituisce il file in modo che contenga dati diversi (ad esempio codice malevolo).*
3. *Se il sistema non ricontrollasse il file, potrebbe elaborare o eseguire il contenuto modificato.*

Inoltre, è stato scelto proprio il calcolo del **checksum** poiché a fronte di notevoli sperimentazioni, è stato sollevato un quesito importante, ovvero i metadati dei file. Solitamente, l'attacco **TOCTOU** si previene andando a evidenziare modifiche provenienti dai **metadati** (*timestamp di creazione, modifica, ecc.*), ma non sempre essi sono **disponibili**: per questo motivo, si è ritenuto necessario procedere con il calcolo del **checksum**.

#### Proposta progettuale

In parallelo, per quanto riguarda il caricamento di una **proposta progettuale**, il file deve essere in formato “txt” e non deve superare la dimensione massima di **dieci MB**. Anche in questo caso,

vengono eseguiti controlli dettagliati sul contenuto del **file**. Viene utilizzato **Tika** per verificare il *content-type* e assicurarsi che il file contenga testo valido, evitando l'inclusione di contenuti non sicuri, come script **JavaScript potenzialmente dannosi**. A tal fine, viene impiegato **Jsoup**, una *libreria HTML parser* che consente di analizzare il contenuto del file e di rimuovere eventuali script o attributi pericolosi come "*onclick*" o "*onload*" garantendo che la proposta caricata non possa essere utilizzata per iniettare codice malevolo nella web app.

```
public static boolean checkProposalFile(HttpServletRequest request, HttpServletResponse response, HttpSession session, Part filePart, ServletContext context, byte[] checksum) {
    byte[] lastChecksum = Encryption.calculateChecksumFromPart(filePart);
    ProposalChecker.checksumControl(request, response, session, checksumOriginal, lastChecksum);
    System.out.println("Checksum 2: " + lastChecksum.toString());
    PasswordManager.clearBytes(lastChecksum);
    // Controlla se il file è stato effettivamente caricato
    if (filePart != null && filePart.getSize() > 0) {
        // Ottieni il nome del file
        String fileName = Paths.get(filePart.getSubmittedFileName()).getFileName().toString();
        System.out.println("fileName: " + fileName);
        // Controlla l'estensione del file
        String fileExtension = fileName.substring(fileName.lastIndexOf(".") + 1).toLowerCase();
        if ("txt".equals(fileExtension)) {
            String realPath = context.getRealPath("/");
            Path filePath = Paths.get(realPath, fileName);
            System.out.println("filePath: " + filePath);
            return true;
        } else {
            // L'estensione del file non è ".txt"
            DisplayMessage.showPanel("Puoi caricare solo file di testo in formato txt!");
        }
    } else {
        // Nessun file caricato
        DisplayMessage.showPanel("Devi caricare una proposta progettuale!");
    }
    // Se si arriva qui, qualcosa è andato storto, restituisci false
    return false;
}
```

### Metodo `checkProposalFile` (classe `ProposalChecker`)

- **Verifica del checksum:** Anche qui, viene calcolato e confrontato il checksum del file caricato con il checksum originale. In caso di discrepanze, il caricamento *fallisce*.
- **Controllo dell'estensione:** Verifica che il file sia di tipo "txt" e restituisce un messaggio di errore all'utente in caso contrario. Il controllo dell'estensione è fondamentale per garantire che i file caricati siano in un formato di testo semplice.
- **Verifica del *content-type*:** Come per le immagini, utilizza *Tika* per verificare che il file caricato sia effettivamente *un file di testo e non altro, come un file eseguibile o binario camuffato*.

```

public static String processFile(HttpServletRequest request, HttpServletResponse response, HttpSession session, Part filePart, byte[] checksumOriginal)

    long maxSizeInBytes = 10 * 1024 * 1024; //Max 10MB
    if (filePart.getSize() > maxSizeInBytes) {
        DisplayMessage.showPanel(
            "Il file supera la dimensione massima consentita. Il file puo' essere massimo di 10 MB");
        return null;
    }

    // Usa try-with-resources per gestire la chiusura dell'input stream
    try (InputStream inputStream = filePart.getInputStream()) {

        byte[] lastChecksum = Encryption.calculateChecksumFromPart(filePart);

        ProposalChecker.checksumControl(request, response, session, checksumOriginal, lastChecksum);

        System.out.println("Checksum 3: " + lastChecksum.toString());
        PasswordManager.clearBytes(lastChecksum);

        // Controlla il content-type con Tika
        Tika tika = new Tika();
        String contentType = tika.detect(inputStream);
        System.out.println(contentType);

        // Rinnoviamo l'InputStream se necessario (necessario perché tika.detect() consuma inputStream)
        if ("text/plain".equals(contentType) || "text/html".equals(contentType)) {
            // Crea un nuovo InputStream per leggere di nuovo i dati
            try (InputStream fileContentStream = filePart.getInputStream()) {
                byte[] contentBytes = ReadByteSecure.readAllBytesSecurely(fileContentStream);

                // Usa Jsoup per rimuovere gli script JavaScript
                Document document = Jsoup.parse(ConvertingType.byteArrayToString(contentBytes)); //aggiunge i tag html
                document.select("script, [type=application/javascript], [type=text/javascript]").remove();
                document.select("[text]").unwrap(); // Rimuove anche il testo all'interno dei tag script

                document.select("[onclick]").removeAttr("onclick");
                document.select("[onload]").removeAttr("onload");

                String cleanedHtml = document.toString();

                checksumOriginal = Encryption.calculateChecksumFile(ConvertingType.stringToByteArray(cleanedHtml));
                System.out.println("Checksum 4: " + checksumOriginal.toString());
                return cleanedHtml;
            }
        } else {
            DisplayMessage.showPanel("Il file contiene del testo non valido!");
        }
    }
}

```

### Metodo processFile (classe ProposalChecker)

- **Controllo delle dimensioni del file:** Limita le dimensioni dei file caricati a un massimo di 10 MB.
- **Verifica del *content-type*:** Come per checkProposalFile, utilizza *Tika* per confermare che il contenuto del file sia di tipo "text/plain" o "text/html".
- **Pulizia del file:** Utilizza *Jsoup* per rimuovere eventuali script JavaScript o attributi HTML pericolosi (*come onclick e onload*) che potrebbero essere utilizzati per eseguire codice malevolo nel browser dell'utente. Questo migliora la sicurezza prevenendo attacchi di **Cross-Site Scripting (XSS)**.

```

public static void checksumControl(HttpServletRequest request, HttpServletResponse response, HttpSession session,
        byte[] checksumOriginalFile, byte[] lastChecksum) throws IOException {

    if(!Arrays.equals(checksumOriginalFile, lastChecksum)){
        if (session != null) {
            session.invalidate();
        }
        Cookie[] cookies = request.getCookies();
        if (cookies != null) {
            for (Cookie cookie : cookies) {
                if ("rememberMe".equals(cookie.getName())) {
                    byte[] cookieByte = Base64.getDecoder().decode(cookie.getValue());
                    if(DeleteTokenDAO.deleteToken(cookieByte)) {
                        // Rimuove il cookie dal browser
                        cookie.setMaxAge(0);
                        cookie.setHttpOnly(true);
                        cookie.setSecure(true);
                        response.addCookie(cookie);
                        response.sendRedirect("userNotLoggedInIndex.jsp");
                        DisplayMessage.showPanel("Logout fato effettuato correttamente!");
                    }
                }
            }
        }
        PasswordManager.clearBytes(lastChecksum);
        PasswordManager.clearBytes(checksumOriginalFile);
        DisplayMessage.showPanel("Non è stato possibile caricare il file della proposta, i file sembrano diversi!");
        response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
    }else {
        DisplayMessage.showPanel("Nessun cambiamento rilevato durante il controllo del checksum della proposta");
    }
}

```

### Metodo checksumControl (classe ProposalChecker)

- **Confronto dei checksum:** Confronta i checksum originale e quello del file caricato. Se non coincidono, il sistema forza il **logout** dell'utente, **invalida** la sessione e rimuove eventuali cookie di **autenticazione**. Questo garantisce che un file non sia stato alterato durante il caricamento.
- **Pulizia della memoria:** Dopo il controllo del checksum, i **dati sensibili** vengono cancellati dalla memoria per evitare che informazioni critiche rimangano in memoria (*es. per prevenire attacchi di memoria*).

Il checksum in questo caso viene calcolato in quattro istanti diversi, ovvero:

- I. Durante l'acquisizione dal form di upload della proposta
- II. Durante il check riguardo il formato del file
- III. Durante il processing del file per la trasformazione in formato HTML e rimozione di script
- IV. Prima del caricamento della proposta nel Database

## Visualizzazione della proposta progettuale

La visualizzazione delle proposte avviene tramite una chiamata **AJAX** al caricamento della pagina, che consente di ottenere in modo **asincrono** la lista delle proposte disponibili. Nel dettaglio:

- **Pagina JSP (sezione di visualizzazione):** La funzione *JavaScript loadProposalList()* viene eseguita non appena la pagina si **carica**, invocando una richiesta **GET** alla servlet *UploadProposalServlet*. La richiesta viene fatta in modalità asincrona utilizzando *jQuery AJAX*, consentendo all'utente di continuare a interagire con la pagina mentre le proposte vengono recuperate.

```
// Funzione per caricare la lista delle proposte
function loadProposalList() {
    // Controlla se la lista delle proposte è già in fase di caricamento
    if (isProposteLoading) {
        console.log("Il caricamento delle proposte è già in corso.");
        return;
    }

    isProposteLoading = true; // Imposta la variabile a true per evitare richieste duplicate

    $.ajax({
        url: 'UploadProposalServlet', // URL dell'API o del server per ottenere le proposte
        method: 'GET',
        dataType: 'json',
        success: function (data) {
            console.log("JSON ricevuto dal server:", data); // Log del JSON ricevuto per il debug

            var proposalList = $('#proposalBannerList'); // Seleziona l'elemento della lista delle proposte
            proposalList.empty(); // Svuota la lista esistente per evitare duplicati

            if (!scrittaAggiunta) {
                proposalList.append('<p><strong>Lista proposte progettuali</strong></p>');
                scrittaAggiunta = false;
            }
        }
    });
}
```

**Servlet UploadProposalServlet:** Nella servlet, il metodo **doGet()** viene utilizzato per ottenere una lista delle proposte da un metodo statico dell'entità **Proposal**. La lista di oggetti **Proposal** viene convertita in formato **JSON** utilizzando la libreria **Gson**, e quindi inviata come risposta alla richiesta **AJAX**. Questo **JSON** contiene le informazioni essenziali di ciascuna proposta, come il nome del file, l'e-mail del proponente e la proposta stessa.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

    List<Proposal> proposte = Proposal.getProposals();

    // Converti la lista in JSON
    String jsonProposte = new Gson().toJson(proposte);

    // Invia la risposta JSON
    response.setContentType("application/json");
    response.setCharacterEncoding("UTF-8");
    response.getWriter().write(jsonProposte);
}
```

**Visualizzazione nella pagina JSP:** Il JSON ricevuto viene elaborato e utilizzato per popolare dinamicamente una lista nella pagina, visualizzando le informazioni di base di ogni proposta sotto forma di link. Questi link contengono **attributi data-\*** che permettono di memorizzare informazioni aggiuntive come il contenuto della proposta in formato HTML.

The screenshot shows a web application interface. At the top, there's a blue header bar with the text "Proposte progettuali" on the left and three buttons on the right: "Homepage", "Aggiorna Lista Proposte", and "Logout". Below the header, the main content area has a title "Benvenuto, Demo@gmail.com!" in bold black font. Underneath the title is a list titled "Lista proposte progettuali" containing several entries. At the bottom of the main content area is a form with a blue button labeled "Carica proposta progettuale" and a file input field with a "Sfoglia..." button. A note below the input field states: "Il file deve essere di formato .txt e pesare massimo 10 MB."

Esempio che riguarda la visualizzazione di tutte le proposte

- Quando l'utente clicca su una **proposta** nella lista visualizzata, il **comportamento** della pagina cambia per mostrare i dettagli della proposta specifica:
  - **Pagina JSP:** Ogni link generato dalla funzione `loadProposalList()` è dotato di un gestore eventi `onClick`. Quando l'utente clicca su una proposta, le informazioni memorizzate negli attributi **data-\* del link** (quali l'e-mail dell'utente e il contenuto della proposta in formato HTML) vengono recuperate e visualizzate all'interno di un elemento div nella pagina.

```
$('.file-link').on('click', function (event) {
  event.preventDefault(); // Previene il comportamento predefinito del link

  // Recupera i valori degli attributi data-* dal link cliccato
  var fileName = $(this).attr('data-fileName');
  var email = $(this).attr('data-email');
  var html = $(this).attr('data-html');

  // Log per verificare i valori recuperati
  console.log("fileName:", fileName);
  console.log("email:", email);
  console.log("html:", html);

  // Verifica se i valori sono mancanti e mostra un messaggio di errore se necessario
  if (fileName === undefined || html === undefined) {
    console.error("Valori mancanti: ", fileName, html);
    alert("Errore: valori mancanti.");
    return;
  }

  // Crea il messaggio da mostrare nell'elemento #bannerContent
  var messaggio = "Stai visualizzando la proposta progettuale <strong>" + fileName + "</strong> di <strong>" + email + "</strong>";
  $('#bannerContent').show();
  $('#bannerContent').html('<p>' + messaggio + '</p>' + html);
  $('#closeBannerButton').show()
});

isProposteLoading = false; // Reimposta la variabile a false per consentire ulteriori richieste
```

*Esempio della pagina principale circa la visualizzazione di una proposta specifica*

## Caricamento di una proposta progettuale

- Il caricamento di una nuova proposta da parte dell'utente avviene attraverso un form **HTML** con il supporto per file **multipart**, che permette di inviare un file al server. Questo file viene poi elaborato dalla *servlet UploadProposalServlet*.
  - **Pagina JSP (sezione di caricamento):** L'utente seleziona un file tramite un input di tipo file, e il file viene inviato tramite il form al server usando **AJAX**. La funzione *uploadFile()* è responsabile dell'invio del file senza ricaricare la pagina:

```

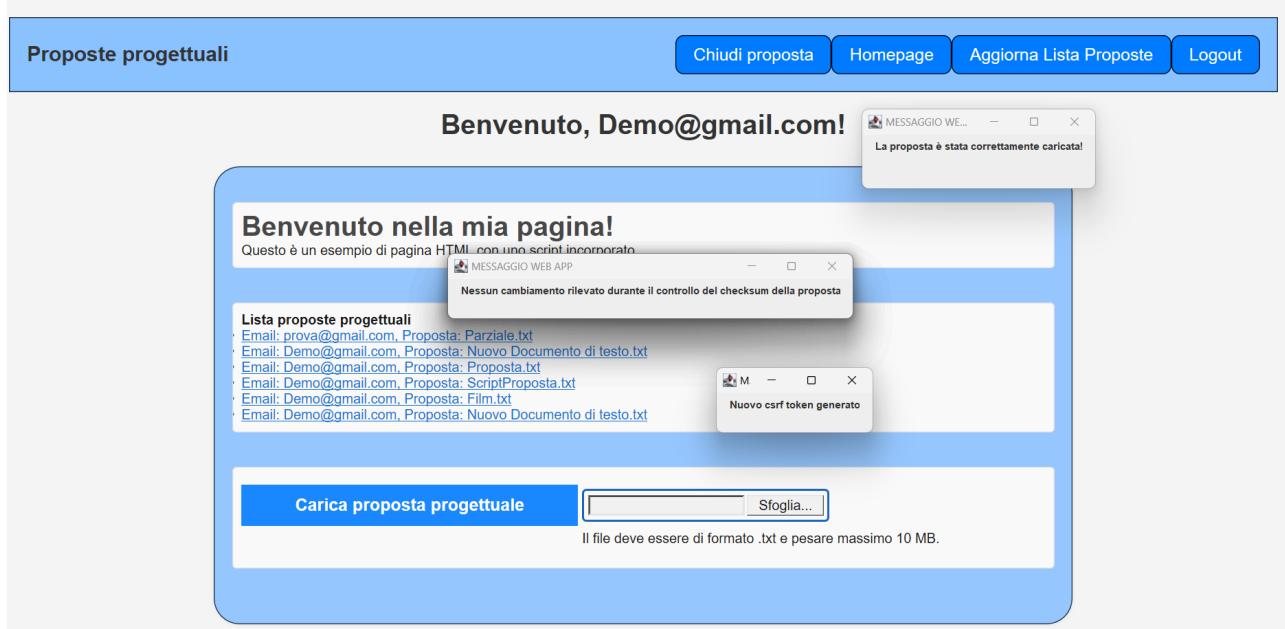
function uploadFile() {
    var formData = new FormData(document.getElementById('uploadForm'));
    $.ajax({
        url : 'UploadProposalServlet', // L'URL della servlet sul server a cui inviare la richiesta
        type : 'POST', // Il tipo di richiesta HTTP, in questo caso POST per inviare dati
        data : formData, //dati da inviare al server
        processData : false, // Impedisce a jQuery di elaborare automaticamente i dati (utile per l'invio di dati binari come i file)
        contentType : false, // Impedisce a jQuery di impostare automaticamente il tipo di contenuto, necessario quando si inviano dati binari
        success : function(data,textStatus,xhr) { // Callback che viene eseguito quando la richiesta ha successo
            // Aggiorna il contenuto del banner con il risultato della risposta AJAX
            var newCsrfToken = xhr.getResponseHeader('X-CSRF-Token');
            if (newCsrfToken) {
                $('#uploadForm [name="csrfToken"]').val(newCsrfToken);
            }
            $('#bannerContent').html(data);
            $('#bannerContent').show();
            $('#uploadButton').hide();
            $('#closeBannerButton').show();
            $('#uploadProposalFile').val('');
        },
        error : function(xhr, status, error) {
            console.error(
                'Errore durante il caricamento del file:',
                status, error);
            if (xhr.status === 401) { // Se ricevi 401, reindirizza l'utente alla pagina di login
                window.location.href = 'userNotLoggedInIndex.jsp';
            }
        }
    });
}

```

**Servlet UploadProposalServlet:** Il metodo *doPost()* della servlet gestisce il **caricamento** del file. Viene prima effettuato un controllo **CSRF** (*Cross-Site Request Forgery*) confrontando il token **CSRF** fornito dall'utente con quello *salvato* nella **sessione**. Per effettuare questo controllo, è presente, inoltre, il campo nascosto nel form (*csrfToken*) contenente il token **CSRF** prelevato dalla sessione. L'idea alla base è che solo l'utente che ha accesso alla sessione possa inviare una richiesta valida con il token corretto. Se il token nella richiesta non fosse quello memorizzato nella sessione, la validazione CSRF fallirebbe, presumibilmente a causa di un attacco.

Inoltre, viene calcolato un checksum sia del file che del suo contenuto per prevenire attacchi **TOCTOU** (*Time-of-Check-to-Time-of-Use*), che potrebbero *modificare il file tra il momento in cui viene verificato e il momento in cui viene effettivamente utilizzato*.

Se il file è valido e il checksum coincide, il contenuto viene processato e salvato nel database. Successivamente, un nuovo token CSRF viene generato e inviato come risposta.



Esempio della pagina principale dopo il corretto caricamento della proposta progettuale

## Gestione Password

La gestione delle password all'interno di tale contesto assume un'importanza rilevante dal punto di vista della **sicurezza**. Una delle tecniche più sicure per gestire le password è l'uso di un **hash crittografico** combinato con un **salt** (*detto anche sale*).

L'**hashing** è una trasformazione **unidirezionale** che prende una stringa di input - *in questo caso la password dell'utente, di lunghezza arbitraria* - e la converte in una stringa di **lunghezza fissa**. Gli **algoritmi di hash** come *SHA-256, bcrypt, Argon2 e PBKDF2* sono progettati per essere resistenti alla decodifica: una volta che la password viene convertita in un **Hash**, è molto difficile risalire alla **password originale**.

Tuttavia, c'è un problema intrinseco nel semplice hashing di password: le **password uguali** producono **hash uguali**. Questo espone il sistema ad **attacchi di dizionario** o **rainbow table**, dove un attaccante può pre-computare gli **hash** per password comuni e confrontarli con quelli presenti **nel database**.

Il **salt** è una sequenza di dati casuali che viene aggiunta alla **password** prima che l'hash venga **calcolato**. Questo migliora significativamente la sicurezza dell'hashing delle password:

- **Aggiunta di Unicità:** Anche se due utenti scelgono la stessa password, i loro hash saranno diversi grazie ai sali differenti. Ad esempio, se la password "password123" viene inserita da due utenti, con due sali distinti ("abc123" e "xyz789"), gli hash risultanti saranno diversi.
- **Difesa contro gli attacchi pre-computati:** Con l'uso di un salt unico per ogni password, un attaccante *non può usare rainbow table pre-calcolati*, poiché l'hash non dipende solo dalla password, ma anche dal salt.

Il **sale** deve essere:

- **Unico:** Deve essere generato in modo casuale per ogni utente.
- **Lungo e complesso:** Per garantire che l'aggiunta del salt renda inefficaci gli attacchi con rainbow table.

È stata anche valutata la possibilità di cifrare il **sale**, giungendo però alla conclusione che trattasi di un'operazione **superflua**. Il **sale**, pur essendo parte della combinazione che produce l'hash, non è considerato **segreto**. Il suo scopo non è quello di essere **una chiave segreta**, ma di aggiungere *casualità e unicità alla password prima dell'hashing*. La sicurezza non deriva dal fatto che il salt sia segreto, ma dal fatto che il salt rende impossibile l'uso di attacchi pre-computati.

## Applicazione Salt e Hash

Nel caso della **Web App**, la classe **PasswordManager** è responsabile di tutte le azioni necessarie per la corretta gestione delle **Password**: la funzione sottostante (*concatenateAndHash*), si occupa di concatenare i **due array di byte** e di effettuare il calcolo **dell'hash** (*utilizzando lo standard NIST odierno SHA-256, ritenuto lo standard di facto*).

```

public static byte[] concatenateAndHash(byte[] password, byte[] salt) {
    try {
        byte[] concatenatedData = new byte[password.length + salt.length];

        System.arraycopy(password, 0, concatenatedData, 0, password.length);

        System.arraycopy(salt, 0, concatenatedData, password.length, salt.length);

        MessageDigest digest = MessageDigest.getInstance("SHA-256");

        return digest.digest(concatenatedData);

    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
        return null;
    }
}

```

Per la **generazione del sale** invece (*un'array di 16 byte*), si utilizza la funzione `generateRandomBytes()` che utilizza un generatore di **byte crittograficamente sicuro** attraverso la classe **Secure Random**.

```

//Generazione sicura di byte
public static byte[] generateRandomBytes(int saltLengths) {

    SecureRandom secureRandom = new SecureRandom();
    byte[] salt = new byte[saltLengths];
    secureRandom.nextBytes(salt);

    return salt;
}

```

L'utilizzo di un hash della password combinato con un **salt** offre numerosi vantaggi in termini di sicurezza:

- **Eliminazione della memorizzazione delle password in chiaro:** Invece di memorizzare le password degli utenti direttamente nel **database** (*che rappresenta una grave vulnerabilità*), viene memorizzato solo **l'hash** della password generato insieme a un **salt** unico. In caso di **violazione del database**, un attaccante non avrà accesso alle password reali, ma solo agli **hash**.
- **Protezione senza la necessità di decifrare o cifrare password:** Poiché l'hash è un processo **unidirezionale**, non c'è bisogno di **decrittografare** nulla durante la fase di **verifica**. Quando l'utente tenta di eseguire il *login*, la **password** inserita viene nuovamente *sottoposta all'hashing con lo stesso salt, e solo l'hash risultante viene confrontato con quello memorizzato nel database*. Questo **elimina** la necessità di gestire chiavi di **decrittazione**, riducendo i **rischi** legati alla gestione delle chiavi stesse.
- **Resistenza agli attacchi pre-computati:** L'uso del salt impedisce attacchi basati su **rainbow table** (*che sono tabelle pre-calcolate di hash comuni*) poiché ogni combinazione di password

e salt produce un **hash unico**. Anche se due utenti usano la stessa password, gli hash memorizzati saranno **diversi** grazie al salt.

- **Scalabilità e compatibilità con sistemi moderni:** Il sistema di hash con salt è ampiamente adottato e compatibile con la maggior parte dei sistemi di autenticazione moderni. Ciò garantisce che la sicurezza possa essere scalata anche per gestire grandi quantità di utenti senza dover compromettere la protezione dei dati sensibili.

## Fase di registrazione

Prima di **registrare** un utente, è fondamentale verificare che la **password** soddisfi i requisiti minimi di **sicurezza** che dovrebbero essere definiti in base all'importanza **dell'asset** da proteggere, come raccomandato anche dal **NIST**. Anche se il sistema implementa solide misure di sicurezza, come *l'hashing con salt e algoritmi di hashing lenti*, una **password debole** compromette significativamente la **sicurezza complessiva**. Una password inadeguata rende vulnerabile anche il sistema più avanzato, poiché riduce drasticamente il livello di protezione e facilita **attacchi**, come quelli a forza bruta o di guessing. Di conseguenza, l'**utente gioca un ruolo chiave nella protezione dei propri dati**, e l'utilizzo di password robuste è essenziale per rendere effettive le misure di sicurezza adottate dal sistema.

```
//Controllo sicurezza password
public static boolean isStrongPassword(byte[] password) {

    final char[] specialCharacters = {'!', '@', '#', '$', '%', '^', '&', '.', '*', '(', ')', '+', '_', '+', '=', '<', '>', '?', '.'};
    boolean valueLength = password.length >= 8;
    boolean hasUpperCase = false;
    boolean hasLowerCase = false;
    boolean hasDigit = false;
    boolean hasSpecialChar = false;

    for (byte b : password) {
        char c = (char) b;

        if (Character.isUpperCase(c)) {
            hasUpperCase = true;
        }

        if (Character.isLowerCase(c)) {
            hasLowerCase = true;
        }

        if (Character.isDigit(c)) {
            hasDigit = true;
        }

        for (char specialChar: specialCharacters) {
            if(c==specialChar) {
                hasSpecialChar=true;
            }
        }
    }
}
```

Il metodo *isStrongPassword* della classe **PasswordManager** si occupa quindi di verificare correttamente i *requisiti di Sicurezza*, ovvero:

- *Almeno 8 caratteri*
- *Una lettera maiuscola*
- *Un Numero*
- *Un carattere speciale*

Durante la fase di **registrazione**, dopo aver controllato correttamente che la password rispettasse i requisiti minimi di sicurezza, verrà effettuata **l'archiviazione** delle **credenziali** all'interno del **Database**.

```
@Immutable
public final class RegistrationDAO {

    public static boolean userRegistration(String email, byte[] password, byte[] salt, Part filePart, byte[] checksumOriginal)
        throws IOException {
        try {
            // Inizializza il driver per comunicare con il DB
            Class.forName("com.mysql.cj.jdbc.Driver");
            try(Connection con_write = DatabaseConnection.getConnectionWrite();
                Connection con_read = DatabaseConnection.getConnectionRead()){

                // Verifica se l'utente esiste già
                if (UserAlreadyExistDAO.userAlreadyExists(con_read, email) > 0) {
                    DisplayMessage.showPanel("Utente già registrato! Email già presente");
                    return false;
                }

                // registrazione nella tabella user
                try(PreparedStatement ps = con_write.prepareStatement(DatabaseQuery.registrationUserQuery())) {
                    ps.setString(1, email);
                    ps.setBytes(2, password);

                    try(InputStream fileContent = filePart.getInputStream()){
                        byte[] lastChecksum = Encryption.calculateChecksumFromPart(filePart);

                        if(!Arrays.equals(checksumOriginal, lastChecksum)) {
                            DisplayMessage.showPanel("Non è stato possibile terminare la registrazione, le immagini profilo non risultano uguali!");
                            return false;
                        }
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
            DisplayMessage.showPanel("Occorso un errore durante la registrazione, riprovare più tardi");
            return false;
        }
    }
}
```

Difatti, dopo aver correttamente verificato che non esista nessun altro account registrato con la stessa **mail**, si procederà alla verifica del **checksum** dell'immagine di profilo. ([TOCTOU](#), visto in precedenza).

Una volta che l'utente sarà stato correttamente registrato all'interno della tabella **user** con **email** e **password** (Hash(password+salt)) si procederà alla registrazione del **salt** all'interno della tabella **salt**.

In questo caso verrà utilizzato **l'id auto incrementale** generato dalla tabella **user** come chiave esterna per registrare il **salt** nella tabella **salt** che successivamente, verrà utilizzato per **confermare** l'identità dell'utente durante la fase di login.

```
int rowsAffected = ps.executeUpdate();

// Imposta lo stato a true se almeno una riga è stata aggiornata
boolean status = (rowsAffected > 0);

// se effettuato correttamente, effettua la query per ricavare l'id dell'utente registrato
if (status) {
    int id_user=TakeUserIdDAO.takeUserId(con_read, email);

    // una volta individuato l'id, registra tramite l'id il salt nella tabella apposita

    if(id_user>0) {
        return (RegistrationUserSaltDAO.registrationUserSalt(con_write, id_user, salt) > 0); // Restituisce true se anche la seconda query ha avuto successo
    }
    else {
        DisplayMessage.showPanel("ID Non valido. Riprovare con un mail corretta. Sale non registrato!");
    }
}

} else {
    DisplayMessage.showPanel("Non è stato possibile terminare la registrazione. Riprovare!");
}
```

```
@Immutable
final class RegistrationUserSaltDAO {
    static int registrationUserSalt(Connection con_write, int id_user, byte[] salt) throws SQLException {
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");
            try (PreparedStatement psSalt = con_write.prepareStatement(DatabaseQuery.registrationUserSaltQuery())){
                psSalt.setInt(1, id_user);

                psSalt.setBytes(2, salt);

                return psSalt.executeUpdate();
            }
        } catch(ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }
        return 0;
    }
}
```

Esempio della pagina DAO *RegistrationUserSaltDAO*, durante l'archiviazione del sale nella tabella salt, utilizzando l'id dell'utente come chiave.

## Fase di Login

Durante la fase di **login**, il meccanismo risulta invariato.

Inizialmente verrà utilizzata la **password** inserita **dall'utente** proveniente dalla richiesta **POST** - opportunamente conservata in un'array di byte, per permettere la successiva sostituzione con byte uguali a zero - dopodiché attraverso la **mail**, verrà recuperato il **sale** che verrà concatenato alla **password inserita** – in questo caso sarà necessario effettuare una join nel Database, per individuare attraverso la **mail**, l'**id corretto** - e successivamente verrà calcolato l'**hash**.

Se il risultato proveniente dalla funzione **Hash** è il medesimo rispetto ai **byte** archiviati all'interno del **Database**, allora l'**accesso** è consentito.

```
@Immutable
public final class LoginDAO {
    public static boolean isUserValid(String email, byte[] password) {
        boolean status = false;

        try {
            Class.forName("com.mysql.cj.jdbc.Driver");

            try( Connection con_read = DatabaseConnection.getConnectionRead()){
                int id_user=TakeUserIdDAO.takeUserId(con_read, email);

                if(id_user>0) {
                    Blob userSaltBlob = TakeUserSaltDAO.takeUserSalt(con_read, id_user);

                    if (userSaltBlob != null) {
                        byte[] sale = userSaltBlob.getBytes(1, (int) userSaltBlob.length());
                        byte[] newPassword = PasswordManager.concatenateAndHash(password, sale);

                        try(PreparedStatement psUser = con_read.prepareStatement(DatabaseQuery.getSelectUserQuery())){
                            psUser.setString(1, email);
                            psUser.setBytes(2, newPassword);

                            try(ResultSet rsUser = psUser.executeQuery()){
                                status = rsUser.next();

                                if (status) {
                                    System.out.println("Utente trovato");
                                } else {
                                    DisplayMessage.showPanel("Errore nell'inserimento dei dati dell'utente. Riprova"); //VALUTARE DI ELIMINARE?
                                }
                            }
                        }
                    }
                }
            }
        }
```

In entrambi i casi, ogni **array** quindi, viene ripulito attraverso il metodo *PasswordManager.clearBytes()*.

## Minimizzazione scope variabili

La minimizzazione dello **scope** delle **variabili** rappresenta una pratica fondamentale per ridurre al minimo l'**esposizione** di dati sensibili e migliorare l'**integrità** e la sicurezza del codice. Definire

variabili con lo scope più ristretto possibile, comporta il vantaggio di **limitare** la visibilità e la durata in memoria dei dati sensibili, riducendo il rischio che tali informazioni vengano accidentalmente o intenzionalmente **esposte**.

Il controllo preciso dell'ambito delle variabili contribuisce, inoltre, a contenere l'impatto di eventuali **vulnerabilità** o attacchi, poiché un attaccante avrebbe un accesso **limitato** a tali variabili. In ambienti altamente sensibili, come nelle operazioni **crittografiche**, minimizzare lo **scope** e la durata in memoria delle chiavi e dei vettori di inizializzazione è **cruciale** per evitare che rimangano accessibili inutilmente, aumentando il rischio di compromissione.

```
public final class Encryption {

    //Cifratura utilizzando AES in modalità CBC
    public static byte[] encrypt(byte[] data) throws Exception {
        char[] AES_KEY=Encryption.readAesKey();
        SecretKey key = Encryption.getSecretKey(AES_KEY);

        if (key != null) {
            char [] AES_IV = Encryption.readAES_IV();
            Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
            cipher.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(ConvertingType.charToBytes(AES_IV)));
            Arrays.fill(AES_KEY, '\0');
            Arrays.fill(AES_IV, '\0');
            return cipher.doFinal(data);
        } else {
            Arrays.fill(AES_KEY, '\0');
            System.out.println("Chiave non valida.");
            return null;
        }
    }
}
```

In questo esempio, le variabili **AES\_KEY** e **AES\_IV**, che contengono rispettivamente la chiave di *cifratura* e il *vettore di inizializzazione*, sono utilizzate solo all'interno di un ambito molto ristretto, all'interno del blocco **if**. Una volta completato l'uso di queste variabili, esse vengono immediatamente sovrascritte con valori nulli (`\0`), **minimizzando** il rischio di esposizione *accidentale*. Questo approccio riduce la possibilità che informazioni sensibili **rimangano in memoria** e possano essere accessibili da altre parti del codice o da potenziali attaccanti che sfruttano eventuali vulnerabilità.

## Accessibilità delle classi

Nel contesto della programmazione orientata agli oggetti e, in particolare, nello sviluppo Java, l'**accessibilità** delle classi gioca un ruolo fondamentale nella gestione della **modularità** e della sicurezza del codice. L'accesso alle classi deve essere progettato in modo tale da garantire che solo le componenti del **sistema** che necessitano di **interagire** con determinati metodi e dati possano farlo, limitando l'esposizione di **informazioni** o funzionalità non necessarie al resto del codice.

L'organizzazione delle classi nei package consente di strutturare il codice in maniera **logica** e **gerarchica**, rendendo possibile una gestione più **efficace** della visibilità delle classi e delle risorse.

Tutte quante le classi – tranne in specifici casi – sono state dichiarate come **pubbliche**, poiché i package sono progettati per riflettere **moduli** funzionalmente **distinti** del sistema, ciascuno con una responsabilità specifica e con delle dipendenze ben definite verso gli altri moduli, quindi *prediligendo alta coesione e basso accoppiamento*. Ogni classe, quindi, seppur essendo pubblica, espone soltanto alcuni **metodi**, definendo quelli utilizzati **internamente** come **private**, ovvero accessibili solo dall'interno. Allo stesso modo, tutte quante le **variabili** interne sono state dichiarate **private**.

```

private byte[] generateFinalToken(byte[] token) {
    return Base64.getEncoder().encode(PasswordManager.concatenateAndHash(this.byte_encryptedEmail, token));
}

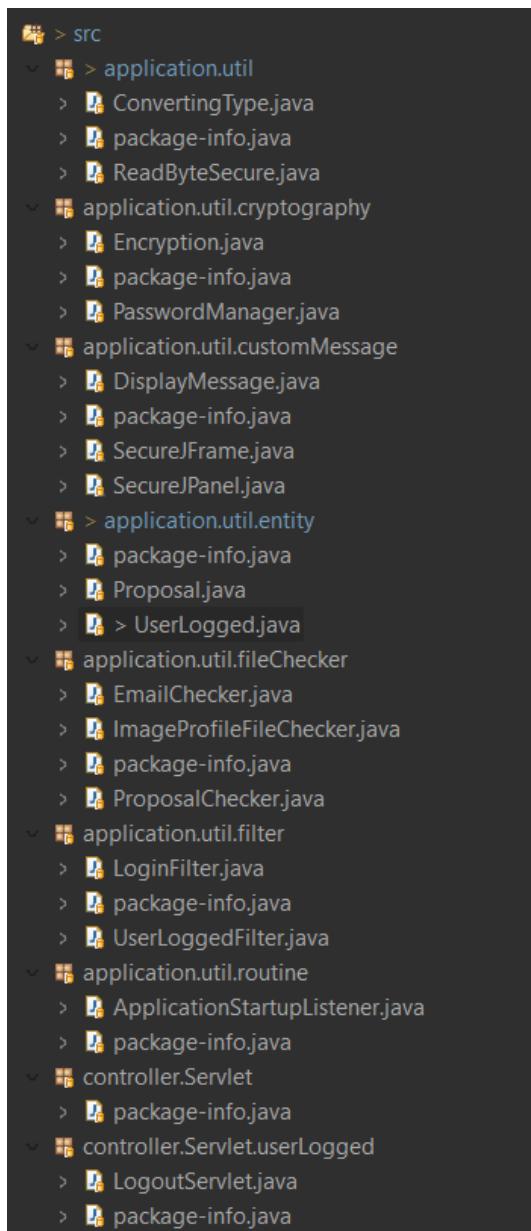
private byte[] calculateTimeStamp(byte[] timestamp) {
    return PasswordManager.concatenateAndHash(timestamp, PasswordManager.generateRandomBytes(timestamp.length));
}

public byte[] getByte_encryptedEmail() {
    return byte_encryptedEmail;
}

```

Nella classe **UserLogged**, per esempio, non vengono esposti i metodi utilizzati per il calcolo del *cookie*, *del timestamp o ancora del token CSRF*, ma bensì vengono esposti soltanto i metodi per ottenere tali valori.

Le classi sono state separate in vari **package** sotto un albero comune come *application.util*, *application.util.cryptography*, *application.util.fileChecker*, *controller.Servlet.userLogged*, ecc.



In questo caso è stata delineata chiaramente la **responsabilità** di ciascun **package** e le relazioni tra di loro. Dichiarare le classi come pubbliche in questo contesto assicura che i diversi **moduli** possano accedere ai servizi offerti dalle altre classi quando necessario e assicurano inoltre:

1. **Modularità e riusabilità:** Definire classi **pubbliche** permette di rendere disponibili determinate *funzionalità* e componenti di un modulo per essere utilizzati in altri moduli o package. In questo caso, è stato necessario rendere le classi sotto **application.util** pubbliche affinché siano accessibili da altre parti del sistema, come le *servlet* o altri componenti dell'applicazione. *Ad esempio, i metodi di crittografia o conversione contenuti in Encryption.java o ConvertingType.java devono probabilmente essere usati anche in controller come UploadProposalServlet.*
2. **Isolamento delle responsabilità:** La struttura dei package serve a mantenere una chiara **distinzione** tra le diverse responsabilità del sistema, il che riduce il rischio di interazioni **indesiderate** tra classi. Pur avendo classi pubbliche, l'organizzazione rigorosa in package consente di mantenere il controllo sul flusso delle informazioni e sull'accesso alle funzionalità, rispettando il principio di segregazione delle responsabilità.
3. **Facilità di manutenzione:** Rendere pubbliche tutte le classi nei vari package permette sviluppare e mantenere il sistema con maggiore flessibilità, senza dover continuamente regolare i modificatori di accesso quando le relazioni tra moduli cambiano o evolvono nel tempo.

Nel caso della classe **UserAlreadyExistDAO** invece – *e quindi, del metodo userAlreadyExists* – ci troviamo di fronte ad una **classe non accessibile all'esterno**, ma accessibile soltanto alle classi contenute nello stesso **package** con una visibilità *package-private*. Questa scelta progettuale deriva dal fatto che tale **query** dovrà essere eseguita soltanto nella fase di **registrazione** dal **DAO** specifico che si occuperà di verificare direttamente la presenza di un utente precedentemente registrato, e non direttamente dalla **Servlet**.

```

1 package model.Dao.registration;
2
3+ import java.sql.Connection;
4+ @Immutable
5 final class UserAlreadyExistDAO {
6     static int userAlreadyExists(Connection con_read, String email) throws SQLException {
7
8         try {
9             Class.forName("com.mysql.cj.jdbc.Driver");
10            try (PreparedStatement ps = con_read.prepareStatement(DatabaseQuery.userAlreadyExist())) {
11                ps.setString(1, email);
12                try (ResultSet rs = ps.executeQuery()) {
13                    if (rs.next()) {
14                        return rs.getInt("count");
15                    }
16                }
17            }
18        } catch(ClassNotFoundException|SQLException e) {
19            DisplayMessage.showPanel("Si è verificato un problema di connessione!");
20            e.printStackTrace();
21        }
22        return 0;
23    }
24 }
25
26
27 }
```

## Feedback dei risultati all'utente

Nel contesto dello sviluppo di applicazioni, fornire un **feedback** appropriato e tempestivo agli utenti è una **componente** essenziale per migliorare l'esperienza utente e garantire la comprensione dello stato del sistema. Il feedback, che può assumere la forma di messaggi informativi, avvisi di errore, o conferme di azioni, aiuta gli utenti a capire meglio l'efficacia delle loro interazioni con il sistema. Nel caso della **sudetta Web App**, è stato scelto come approccio quello di mostrare delle **finestre pop-up** utilizzando Java Swing<sup>3</sup>. *Una comunicazione chiara riduce la frustrazione e garantisce che l'utente sia sempre consapevole di ciò che accade durante l'interazione con l'applicazione.*

Le classi che fanno parte del package ***application.util.customMessage***, definiscono un sistema di feedback visivo all'interno di una finestra grafica con **Swing**, garantendo però la sicurezza e l'integrità delle componenti.

```
public final class DisplayMessage{

    public static void showPanel(String message) {
        SwingUtilities.invokeLater(() -> {
            // Crea una finestra e aggiungi il pannello
            SecureJFrame frame = new SecureJFrame("MESSAGGIO WEB APP");
            frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

            SecureJPanel panel = new SecureJPanel(message);
            frame.getContentPane().add(panel);

            frame.pack();
            frame.setLocationRelativeTo(null);
            frame.setAlwaysOnTop(true);

            frame.setVisible(true);

            frame.addWindowListener(new java.awt.event.WindowAdapter() {
                @Override
                public void windowClosing(java.awt.event.WindowEvent windowEvent) {
                    frame.setVisible(false);
                }
            });
        });
    }
}
```

La classe ***DisplayMessage*** è la principale responsabile per la *visualizzazione* dei messaggi. Essa utilizza il pattern del "thread-safe singleton" per mostrare i messaggi in modo sicuro e senza conflitti tra **thread**. La sicurezza dei thread viene garantita dall'uso di ***SwingUtilities.invokeLater()***, che assicura che tutte le operazioni relative all'interfaccia **grafica** vengano eseguite nel **thread** dell'Event Dispatch (EDT), necessario per evitare **condizioni di race** nelle applicazioni **Swing**.

La mutabilità delle variabili è ridotta al minimo, poiché ogni finestra (**JFrame**) viene creata e distrutta in modo isolato all'interno del metodo **showPanel**, senza lasciare spazio a possibili manipolazioni.

---

<sup>3</sup> Swing è un **framework** per Java, appartenente alle **Java Foundation Classes** e orientato allo sviluppo di **interfacce grafiche**. Parte delle classi del framework Swing sono implementazioni di widget come **caselle di testo, pulsanti, pannelli e tavole**.

```
@ThreadSafe
final class SecureJFrame extends JFrame {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    SecureJFrame(String title) {
        super(title);
        // Impostazioni di default di chiusura
        setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    }

    // Sovrascrivere setVisible per loggare o bloccare tentativi di manipolazione.
    @Override
    public void setVisible(boolean b) {
        super.setVisible(b);
    }

    // Sovrascrivere dispose per loggare o bloccare tentativi di manipolazione.
    @Override
    public void dispose() {
        super.dispose();
    }

    // Sovrascrivere metodi chiave
    @Override
    public void addWindowListener(WindowListener l) {
        super.addWindowListener(l);
    }
}
```

La classe **SecureJFrame**, che estende **JFrame**, ha il compito di fornire una finestra sicura e immune da tentativi di manipolazione non autorizzata.

Essa sovrascrive metodi chiave come *setVisible*, *dispose* e *addWindowListener* per loggare o impedire eventuali modifiche non autorizzate. Questo garantisce che il comportamento della finestra rimanga sotto controllo anche se altri componenti dell'applicazione tentassero di **manipolarla**, per esempio estendendo la classe, quindi garantendo **l'immutabilità**.

Un aspetto cruciale di questa classe è la decisione di sovrascrivere metodi del ciclo di vita della finestra per garantire la sua integrità. Ad esempio, nel metodo *dispose*, la sovrascrittura permette di intervenire nel momento in cui si cerca di chiudere la finestra, assicurando che non vengano eseguite azioni non desiderate senza una verifica adeguata.

```

@ThreadSafe
final class SecureJPanel extends JPanel {

    private static final long serialVersionUID = 1L;
    private JLabel label;

    SecureJPanel(String message) {
        this.label = new JLabel(message);
        add(label);
    }

    // Sovrascrivere il metodo add per loggare o impedire l'aggiunta di componenti non autorizzati.
    @Override
    public void addNotify() {
        super.addNotify();
    }

    // Sovrascrivere il metodo remove per loggare o impedire la rimozione di componenti.
    @Override
    public void removeNotify() {
        super.removeNotify();
    }

    // Sovrascrivere il metodo getPreferredSize per evitare manipolazioni delle dimensioni del pannello.
    @Override
    public Dimension getPreferredSize() {
        Dimension preferredSize = super.getPreferredSize();
        int width = (int) preferredSize.getWidth() + 20;
        int height = (int) preferredSize.getHeight() + 20;
        return new Dimension(width, height);
    }
}

```

Allo stesso modo, la classe **SecureJPanel** estende **JPanel** per creare un pannello sicuro all'interno della finestra. Anche qui, i metodi chiave come *addNotify*, *removeNotify* e *getPreferredSize* vengono sovrascritti per impedire modifiche non autorizzate alle componenti grafiche o alle dimensioni del pannello. Ad esempio, nel metodo *getPreferredSize*, viene imposto un controllo sulle dimensioni del pannello, aumentando leggermente larghezza e altezza rispetto alle dimensioni suggerite dalla superclasse, prevenendo manipolazioni non desiderate da parte di utenti malintenzionati.

## Implementazione feedback visivi

La classe **DisplayMessage** è l'unica che viene effettivamente richiamata e utilizzata dal codice per mostrare un messaggio agli utenti. Il suo scopo è quello di creare e visualizzare una finestra con un messaggio di testo all'interno di un pannello, garantendo che questo processo avvenga in modo sicuro e senza rischi di manipolazione o condizioni di competizione tra *thread*.

- **Metodo showPanel:** È il punto centrale della classe. Questo metodo crea e visualizza una finestra (**SecureJFrame**) che contiene un pannello (**SecureJPanel**) con il messaggio passato come parametro.

*Il metodo si occupa di:*

- *Creare una finestra (SecureJFrame) con un titolo.*
- *Aggiungere un pannello (SecureJPanel) contenente il messaggio al suo interno.*
- *Impostare le opzioni della finestra, come la possibilità di chiuderla e la sua visibilità.*
- *Aggiungere un listener per gestire la chiusura della finestra in modo sicuro.*

**SecureJFrame** estende la classe **JFrame** di Swing e fornisce una finestra sicura per visualizzare i messaggi all'utente.

- **Gestione sicura della chiusura:** Il metodo *dispose()* assicura che, quando la finestra viene chiusa, tutte le risorse vengano gestite correttamente, evitando che la finestra rimanga visibile o che ci siano altri problemi legati alla sua chiusura. Questo è fondamentale per prevenire attacchi legati alla gestione della finestra.

La classe `SecureJFrame` è progettata per essere utilizzata solo internamente da `DisplayMessage`, rendendo impossibile per altri componenti dell'applicazione manipolare o alterare le finestre di messaggio in modo non controllato.

**Secure JPanel estende JPanel di Swing** e fornisce un pannello sicuro che visualizza il messaggio all'interno della finestra.

- **Aggiunta del messaggio:** Il pannello viene inizializzato con un'etichetta (**JLabel**) contenente il messaggio di testo. Questo messaggio viene visualizzato al centro del pannello.
- **Gestione delle dimensioni:** Il metodo `getPreferredSize()` è stato sovrascritto per impedire manipolazioni indebite delle dimensioni del pannello. Anche se il layout o l'interfaccia tenta di ridimensionare il pannello, questa sovrascrittura assicura che le dimensioni vengano controllate e mantenute in modo sicuro.

L'architettura è stata progettata per minimizzare i rischi derivanti da eventuali manipolazioni esterne, garantendo che le finestre e i pannelli utilizzati per visualizzare messaggi non possano essere alterati in modo non autorizzato.

Le classi ***DisplayMessage***, ***SecureJFrame*** e ***Secure JPanel*** rappresentano un esempio efficace di come costruire un sistema di feedback per gli utenti che sia sicuro e conforme alle best practices di progettazione orientata agli oggetti. L'uso delle annotazioni **@ThreadSafe** - come vedremo successivamente - e la dichiarazione delle classi come **final** minimizza i rischi associati alla **mutabilità** e alla **concorrenza**, garantendo che le finestre create per visualizzare i messaggi non possano essere alterate in maniera indebita.

## Thread Safety

L'annotazione del codice sorgente rappresenta un meccanismo attraverso il quale è possibile associare **metadati** a un elemento del programma, rendendoli accessibili al **compilatore**, **agli strumenti di analisi**, **ai debugger** o **alla Java Virtual Machine (JVM)**.

Esistono diverse annotazioni specifiche per documentare la sicurezza dei **thread**. In particolare, il framework **JCIP** (*Java Concurrency in Practice*) fornisce tre annotazioni a livello di classe per descrivere l'intento progettuale del programmatore in relazione alla sicurezza dei thread.

L'annotazione **@ThreadSafe** viene applicata a una classe per indicare che essa è sicura rispetto all'esecuzione concorrente (*thread-safe*). Questo implica che nessuna sequenza di accessi, che includa letture e scritture su campi pubblici o chiamate a metodi pubblici, può lasciare l'oggetto in uno stato inconsistente o incoerente.

L'annotazione **@Immutable**, invece, viene utilizzata per identificare classi immutabili. Gli oggetti immutabili sono intrinsecamente thread-safe; una volta creati, possono essere condivisi in modo sicuro tra più thread senza richiedere ulteriori misure di sincronizzazione, grazie alla loro natura immodificabile dopo la costruzione.

```

package model.Dao.db;

import java.io.IOException;
@Immutable
public final class DatabaseConnection {

    private final static String URL;
    private final static String USERNAME_READ;
    private final static String PASSWORD_READ;

    private final static String USERNAME_WRITE;
    private final static String PASSWORD_WRITE;

    private final static String USERNAME_DELETE;
    private final static String PASSWORD_DELETE;

    static {
        try (InputStream input = Thread.currentThread().getContextClassLoader().getResourceAsStream("config.ini")) {
            Properties prop = new Properties();
            prop.load(input);

            URL = prop.getProperty("db.url");
            USERNAME_READ = prop.getProperty("db.username_read");
            PASSWORD_READ = prop.getProperty("db.password_read");

            USERNAME_WRITE = prop.getProperty("db.username_write");
            PASSWORD_WRITE = prop.getProperty("db.password_write");

            USERNAME_DELETE = prop.getProperty("db.username_delete");
            PASSWORD_DELETE = prop.getProperty("db.password_delete");
        }

        /**
         * Log per il debug
         System.out.println("Configurazione caricata correttamente.");
        */
    }
}

```

Esempio della classe Database Connection definita immutabile

**Thread-safety** identifica una porzione di codice che può essere eseguita correttamente e prevedibilmente da più thread contemporaneamente.

Un oggetto o metodo è *thread-safe* se più thread possono interagire con esso simultaneamente senza causare errori, condizioni di gara o comportamenti imprevisti.

La maggioranza delle classi, sono state dichiarate **final** a causa di scelte progettuali, poiché non necessitano ulteriori estensioni a causa di un **alta coesione**; essendo **final** risultano quindi **immutabili**, ovvero un oggetto il cui stato non può essere cambiato dopo la sua creazione. Gli oggetti immutabili sono **intrinsecamente thread-safe** perché non possono essere modificati dopo la costruzione. Questo elimina i problemi di concorrenza, poiché più thread possono accedere contemporaneamente agli stessi dati senza rischi di modifica.

Di seguito alcune delle classi considerate immutabili e alcune considerate Thread Safe.

Classe	Immutabile (@Immutable)	Thread-safe (@ThreadSafe)	Considerazioni
<i>ConvertingType</i>	No	Sì	La classe ha solo metodi statici e non ha variabili condivise.
<i>Encryption</i>	Sì	Sì	Le risorse critiche come Cipher e MessageDigest sono locali e non condivise. È sicura per i thread.
<i>PasswordManager</i>	No	Condizionale	Sicura tranne per clearBytes, che potrebbe causare problemi se l'array è condiviso tra thread.

<i>DisplayMessage</i>	No	Solo se usato in EDT	Non è immutabile, poiché manipola oggetti Swing mutabili. La GUI deve essere gestita nel thread EDT.
<i>SecureJFrame</i>	No	Solo se usato in EDT	Estende una classe mutabile (JFrame). Deve essere usata solo nel thread EDT per essere sicura.
<i>Secure JPanel</i>	No	Solo se usato in EDT	Simile a SecureJFrame, deve essere utilizzata solo nel thread EDT per essere thread-safe.
<i>Proposal</i>	Sì	Sì	È immutabile. Tuttavia, i metodi statici che interagiscono con risorse esterne devono gestire la concorrenza.
<i>UserLogged</i>	No	No	Non è immutabile e presenta campi che potrebbero essere modificati. Potrebbe richiedere sincronizzazione.
<i>ProposalChecker</i>	No	Sì	I metodi sono statici e non modificano stati interni. Interagisce con oggetti esterni (es. sessioni HTTP) che potrebbero richiedere attenzione alla sicurezza dei thread.
<i>ImageProfileChecker</i>	No	Si	I metodi della classe sono statici e non modificano lo stato della classe stessa. Questo suggerisce che la classe è thread-safe poiché non c'è stato condiviso tra i thread.
<i>LoginFilter</i>	No	Condizionale	Non ha variabili di istanza ma interagisce con

			oggetti come sessioni HTTP, che non sono thread-safe. La sicurezza dipende dal contesto e dalla gestione delle risorse.
<i>UserLoggedFilter</i>	No	Condizionale	Simile a LoginFilter, la classe è priva di stato interno, ma gestisce sessioni e cookie che possono essere condivisi tra più thread.
<i>ApplicationStartupListener</i>	No	Sì	Utilizza un esecutore a thread singolo per la gestione sicura dei task. Annotata come <code>@Immutable</code> per indicare che l'oggetto non cambia stato.

In questa tabella troviamo soltanto **alcune** delle classi utilizzate all'interno della **Web App**, poiché le classi **Servlet** sono *intrinsecamente Thread Safe* e come specificato in alcune classi, solitamente trattandosi di risorse condivise (*Es. Cookie*) è necessario gestire autonomamente le risorse stesse.

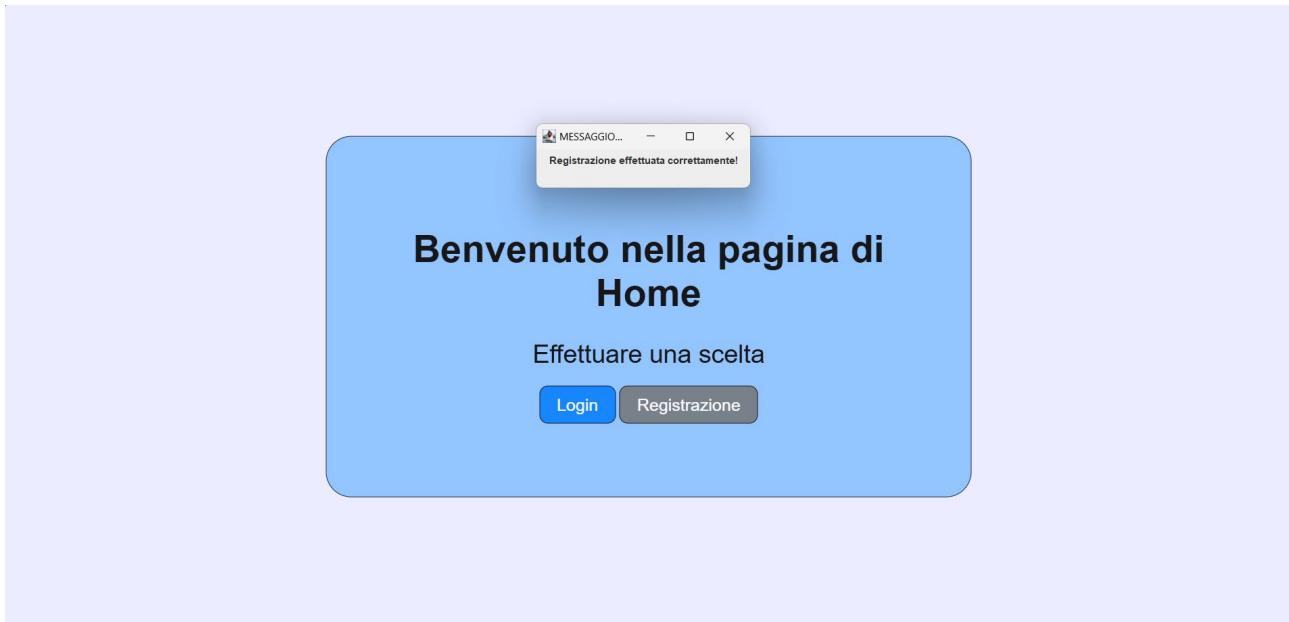
## Analisi Dinamica

Durante l'analisi dinamica, sono state testate tutte le **funzionalità** richieste, analizzando sia esiti negativi che positivi.

### Test d'uso

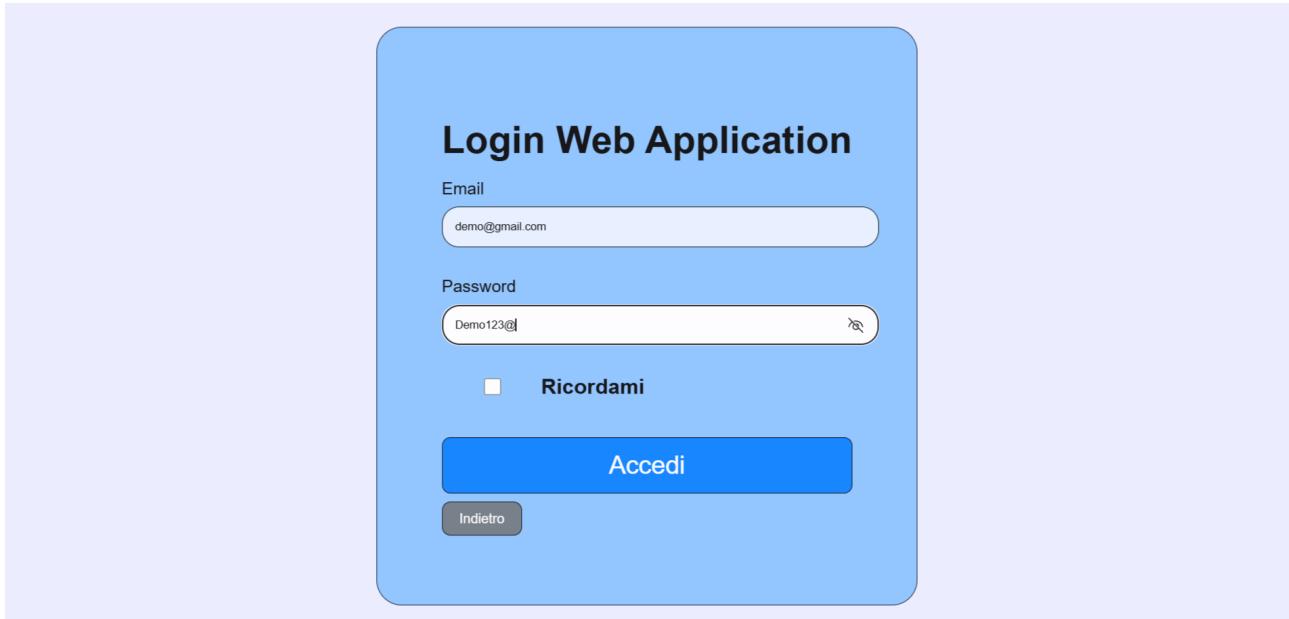
#### Caso d'uso I: Registrazione effettuata correttamente

Descrizione	Pre-condizioni	Post-condizioni
L'utente effettua la registrazione in maniera corretta	Nessuna	<ul style="list-style-type: none"> <li>L'utente viene registrato all'interno del sistema</li> <li>L'utente viene reindirizzato all'Homepage</li> </ul>



## Caso d'uso II: Login effettuato correttamente senza meccanismo Remember me

Descrizione	Pre-condizioni	Post-condizioni
L'utente effettua il login utilizzando le credenziali correttamente registrate senza il meccanismo dei cookie	L'utente deve essere registrato all'interno del sistema	<ul style="list-style-type: none"> <li>L'utente viene reindirizzato alla pagine delle proposte</li> <li>Una nuova sessione viene creata</li> </ul>



**Proposte progettuali**

[Homepage](#) [Aggiorna Lista Proposte](#) [Logout](#)

**Benvenuto, demo@gmail.com!**

### Caso d'uso III: Login effettuato correttamente con meccanismo Remember me

Descrizione	Pre-condizioni	Post-condizioni
L'utente effettua il login utilizzando le credenziali correttamente registrate con il meccanismo dei cookie	L'utente deve essere registrato all'interno del sistema	<ul style="list-style-type: none"> <li>L'utente viene reindirizzato alla pagine delle proposte</li> <li>Il cookie di durata pari ad 1 giorno viene registrato nel sistema</li> <li>Una nuova sessione viene creata</li> </ul>

**Proposte progettuali**

[Homepage](#) [Aggiorna Lista Proposte](#) [Logout](#)

**Benvenuto, demo@gmail.com!**

A screenshot of a web application titled "Proposte progettuali". At the top right are links for "Homepage", "Aggiorna Lista Proposte", and "Logout". Below this, a welcome message "Benvenuto, demo@gmail.com!" is displayed. A modal window titled "Lista proposte progettuali" shows a message "Cookie registrato correttamente!". Below the modal is a file upload input field with the placeholder "Carica proposta progettuale" and a note "Il file deve essere di formato .txt e pesare massimo 10 MB.".

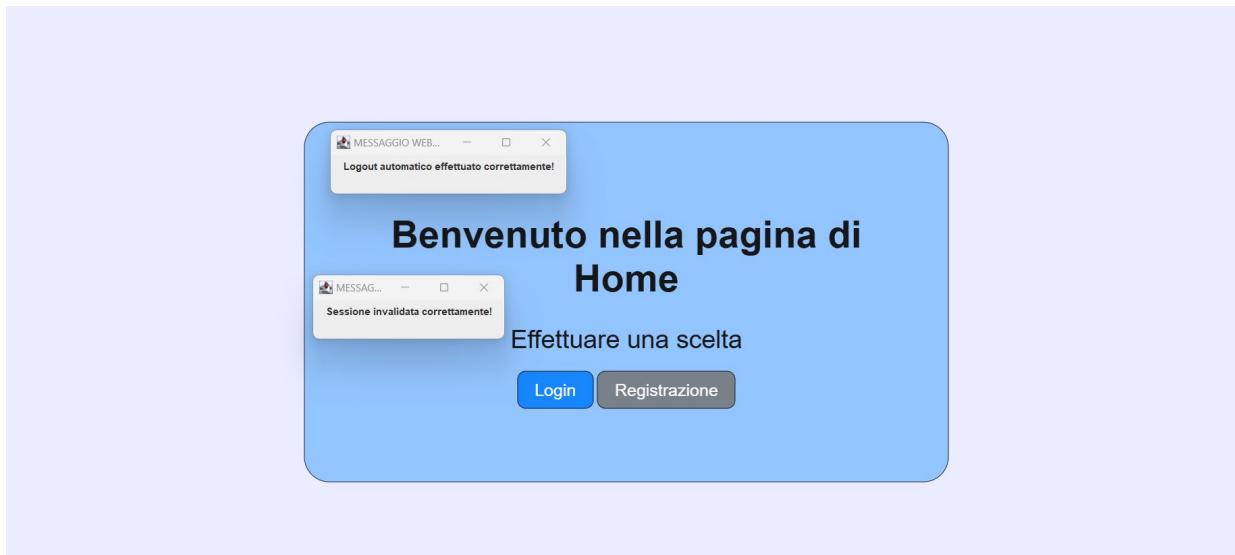
#### Caso d'uso IV: Logout manuale effettuato correttamente

Descrizione	Pre-condizioni	Post-condizioni
L'utente effettua manualmente il logout	<ul style="list-style-type: none"> <li>L'utente deve essere registrato all'interno del sistema</li> <li>L'utente deve essere loggato all'interno del sistema</li> </ul>	<ul style="list-style-type: none"> <li>L'utente viene reindirizzato alla Homepage</li> <li>La sessione viene invalidata</li> <li>I cookie permanenti, se presenti, vengono distrutti all'interno del dispositivo e cancellati nel database</li> </ul>

A screenshot of a web application titled "Benvenuto nella pagina di Home". The main content area says "Effettuare una scelta" with "Login" and "Registrazione" buttons. Three message boxes are overlaid: 1) "Logout manuale effettuato correttamente!" (top left), 2) "Cookie eliminato con successo" (center), and 3) "Logout manuale senza cookie effettuato correttamente!" (bottom right).

## Caso d'uso V: Logout automatico effettuato correttamente

Descrizione	Pre-condizioni	Post-condizioni
L'utente non interagisce per 15 minuti all'interno della Web App	<ul style="list-style-type: none"> <li>L'utente deve essere registrato all'interno del sistema</li> <li>L'utente deve essere loggato all'interno del sistema</li> </ul>	<ul style="list-style-type: none"> <li>L'utente viene reindirizzato alla Homepage</li> <li>La sessione viene invalidata</li> </ul>



## Caso d'uso VI: Proposta caricata in maniera corretta

Descrizione	Pre-condizioni	Post-condizioni
L'utente carica correttamente la proposta nel formato corretto senza introdurre script malevoli	<ul style="list-style-type: none"> <li>L'utente deve essere registrato all'interno del sistema</li> <li>L'utente deve essere loggato all'interno del sistema</li> </ul>	<ul style="list-style-type: none"> <li>L'utente visualizza la proposta correttamente caricata e il contenuto HTML viene renderizzato all'interno della pagina delle proposte</li> <li>La proposta viene registrata all'interno del sistema</li> </ul>

**Proposte progettuali**

[Homepage](#) [Aggiorna Lista Proposte](#) [Logout](#)

**Benvenuto, demo@gmail.com!**

**Lista proposte progettuali**

**Carica proposta progettuale**  Proposta.txt  
Il file deve essere di formato .txt e pesare massimo 10 MB.

**Carica Proposta**

**Proposte progettuali**

[Chiudi proposta](#) [Homepage](#) [Aggiorna Lista Proposte](#) [Logout](#)

**Benvenuto, demo@gmail.com!**

**Proposta Progettuale**

Progetto: Nome del Progetto  
Data: 23 Agosto 2024  
Autore: Nome dell'Autore

**1. Introduzione**  
Questa proposta progettuale delinea le specifiche per il progetto **Nome del Progetto**. L'obiettivo del progetto è di fornire una soluzione innovativa per...

**2. Obiettivi del Progetto**  
Obiettivo 1: Descrizione dell'obiettivo 1.  
Obiettivo 2: Descrizione dell'obiettivo 2.  
Obiettivo 3: Descrizione dell'obiettivo 3.

**3. Descrizione del Progetto**  
Il progetto si articola in diverse fasi, ognuna delle quali ha un insieme specifico di attività e deliverable. Le fasi del progetto includono:  
. Fase 1: Analisi dei requisiti e pianificazione.  
. Fase 2: Progettazione e sviluppo.  
. Fase 3: Testing e valutazione.  
. Fase 4: Implementazione e rilascio.

**4. Risorse Necessarie**  
Per il completamento del progetto, sono necessarie le seguenti risorse:  
Personale: Ruoli e competenze necessarie (es. sviluppatori, designer, project manager).  
Software: Strumenti e piattaforme richieste (es. IDE, sistemi di versionamento).  
Hardware: Requisiti hardware (es. server, dispositivi di test).

**5. Tempistiche e Milestone**  
Il progetto sarà compilato in un periodo di 6 mesi con le seguenti milestone chiave:

**Messaggi**

- MESSAGGIO WE... - X La proposta è stata correttamente caricata!
- MESSA... - X Nuovo csrf token generato

## Caso d'uso VII: L'utente visualizza tutte le proposte progettuali

Descrizione	Pre-condizioni	Post-condizioni
L'utente visualizza tutte le proposte all'interno della pagina delle proposte	<ul style="list-style-type: none"> <li>• L'utente deve essere registrato all'interno del sistema</li> <li>• L'utente deve essere loggato all'interno del sistema</li> </ul>	L'utente visualizza la liste di tutte le proposte caricate all'interno del sistema

**Proposte progettuali**

[Homepage](#) [Aggiorna Lista Proposte](#) [Logout](#)

**Benvenuto, demo@gmail.com!**

**Lista proposte progettuali**

- [Email: demo@gmail.com, Proposta: Proposta.txt](#)

**Carica proposta progettuale**  Nessun file scelto

Il file deve essere di formato .txt e pesare massimo 10 MB.

## Caso d'uso VIII: L'utente visualizza una proposta progettuale specifica

Descrizione	Pre-condizioni	Post-condizioni
L'utente cliccando su una proposta specifica, ne visualizza il contenuto HTML renderizzato, il nome e l'utente che ha caricato la suddetta	<ul style="list-style-type: none"> <li>• L'utente deve essere registrato all'interno del sistema</li> <li>• L'utente deve essere loggato all'interno del sistema</li> <li>• L'utente deve cliccare su una proposta specifica</li> </ul>	L'utente visualizza la proposta nell'apposito banner all'interno della pagina delle proposte

**Proposte progettuali**

[Chiudi proposta](#) [Homepage](#) [Aggiorna Lista Proposte](#) [Logout](#)

**Benvenuto, demo@gmail.com!**

Stai visualizzando la proposta progettuale **Proposta.txt** di **demo@gmail.com**

**Proposta Progettuale**

Progetto: *Nome del Progetto*  
Data: 23 Agosto 2024  
Autore: *Nome dell'Autore*

**1. Introduzione**  
Questa proposta progettuale delinea le specifiche per il progetto *Nome del Progetto*. L'obiettivo del progetto è di fornire una soluzione innovativa per...

**2. Obiettivi del Progetto**  
Obiettivo 1: Descrizione dell'obiettivo 1.  
Obiettivo 2: Descrizione dell'obiettivo 2.  
Obiettivo 3: Descrizione dell'obiettivo 3.

**3. Descrizione del Progetto**  
Il progetto si articola in diverse fasi, ognuna delle quali ha un insieme specifico di attività e deliverable. Le fasi del progetto includono:  
. Fase 1: Analisi dei requisiti e pianificazione.  
. Fase 2: Progettazione e sviluppo.  
. Fase 3: Testing e valutazione.  
. Fase 4: Implementazione e rilascio.

**4. Risorse Necesarie**  
Per il completamento del progetto, sono necessarie le seguenti risorse:  
Personale: Ruoli e competenze necessarie (es. sviluppatori, designer, project manager).  
Software: Strumenti e piattaforme richieste (es. IDE, sistemi di versionamento).  
Hardware: Requisiti hardware (es. server, dispositivi di test).

**5. Tempistiche e Milestone**  
Il progetto sarà completato in un periodo di 6 mesi, con le seguenti milestone chiave:

Milestone	Descrizione	Data di Completamento

## Caso d'uso IX: L'utente non loggato visita la pagina di login con cookie precedentemente registrato senza una sessione inizializzata

Descrizione	Pre-condizioni	Post-condizioni
L'utente visita la pagina di login	<ul style="list-style-type: none"> <li>• L'utente deve essere registrato all'interno del sistema</li> <li>• Utente non loggato</li> <li>• Cookie registrati validi</li> </ul>	<ul style="list-style-type: none"> <li>• Il sistema reindirizza l'utente alla pagina delle proposte</li> <li>• Una nuova sessione viene creata</li> </ul>

The screenshot shows a user interface for managing project proposals. At the top, there's a navigation bar with 'Homepage', 'Aggiorna Lista Proposte', and 'Logout' buttons. The main area displays a welcome message 'Benvenuto, demo@gmail.com!' followed by a file upload section. The upload section includes a button 'Carica proposta progettuale' and a file input field with placeholder 'Scegli il file' and message 'Nessun file scelto'. A note below says 'Il file deve essere di formato .txt e pesare massimo 10 MB.' Below this are two message boxes from 'MESSAGGIO WEB APP'. The first box says 'Cookie ancora valido!' and the second box says 'Cookie esistente!'. This visualizes the system's behavior of accepting an old cookie while creating a new session.

## Caso d'uso X: L'utente loggato tenta di effettuare nuovamente il login dopo aver visitato l'Homepage

Descrizione	Pre-condizioni	Post-condizioni
L'utente visita la pagina di login	<ul style="list-style-type: none"> <li>• Utente loggato</li> <li>• Sessione valida</li> <li>• Cookie non registrati</li> </ul>	<ul style="list-style-type: none"> <li>• Il sistema reindirizza l'utente alla pagina delle proposte</li> </ul>

**Proposte progettuali**

[Homepage](#) [Aggiorna Lista Proposte](#) [Logout](#)

**Benvenuto, demo@gmail.com!**

**Lista proposte progettuali**  
• [Email: demo@gmail.com, Proposta: Proposta.txt](#)

**Carica proposta progettuale**  Nessun file scelto  
Il file deve essere di formato .txt e pesare massimo 10 MB.

## Test d'abuso

### Caso d'uso XI: Registrazione con e-mail già in uso

Descrizione	Pre-condizioni	Post-condizioni
L'utente effettua la registrazione con una mail già registrata	Nessuna	<ul style="list-style-type: none"> <li>Il sistema reindirizza l'utente sulla medesima pagina ripulendo tutti quanti i campi</li> <li>L'utente non viene registrato</li> </ul>

**Registrazione Web Application**

Email

Password   
Minimo 8 caratteri con un carattere speciale, un numero e una Lettera Maiuscula

Reinserisci password

Immagine del profilo (Max 5MB)  Nessun file scelto  
(Formati accettati: PNG,JPG,JPEG)

**Registrati**

[Indietro](#)

**MESSAGGIO...**  
Utente già registrato! Email già presente

**MESSAGGIO...**  
Errore durante la registrazione nel DB!

## Caso d'uso XII: Registrazione con password debole

Descrizione	Pre-condizioni	Post-condizioni
L'utente effettua la registrazione con una password che non rispetta i <u>requisiti minimi di sicurezza</u>	Nessuna	<ul style="list-style-type: none"> <li>Il sistema reindirizza l'utente sulla medesima pagina ripulendo tutti quanti i campi</li> <li>L'utente non viene registrato</li> </ul>

The screenshot shows a registration form titled "Registrazione Web Application". It includes fields for Email (demo@gmail.com), Password (a short string of dots), and Re-enter password (demo1). There is also a file upload field for a profile picture with the placeholder "fotoHacker.png" and a note about accepted formats: "Formati accettati: PNG,JPG,JPEG". A large blue "Registrati" button is at the bottom. Below the form, a small "Indietro" (Back) button is visible.

Below the form, a separate window titled "MESSAGGIO WEB APP" displays the error message: "La password non rispetta i requisiti minimi di sicurezza!" (The password does not meet the minimum security requirements!).

## Caso d'uso XIII: Registrazione con immagine profilo avente estensione non valida

Descrizione	Pre-condizioni	Post-condizioni
L'utente effettua la registrazione caricando un file con estensione non supportata ( <i>non caricando quindi un file di tipo jpeg, png o jpg</i> )	Nessuna	<ul style="list-style-type: none"> <li>Il sistema reindirizza l'utente sulla medesima pagina ripulendo tutti quanti i campi</li> <li>L'utente non viene registrato</li> </ul>

**Registrazione Web Application**

Email  
demo@gmail.com

Password  
.....

Reinserisci password  
.....

Immagine del profilo (Max 5MB)

(Formati accettati: PNG,JPG,JPEG)

**Registrati**

**Indietro**

MESSAG... — ×  
Estensione del file non supportata.

MESSAGGIO WEB APP — ×  
Immagine non valida!

## Caso d'uso XIV: Registrazione con immagine profilo avente estensione valida ma formato non coerente

Descrizione	Pre-condizioni	Post-condizioni
L'utente effettua la registrazione caricando un file con estensione apparentemente supportata ma non valida	Nessuna	<ul style="list-style-type: none"> <li>Il sistema reindirizza l'utente sulla medesima pagina ripulendo tutti quanti i campi</li> <li>L'utente non viene registrato</li> </ul>

**Registrazione Web Application**

Email  
demo@gmail.com

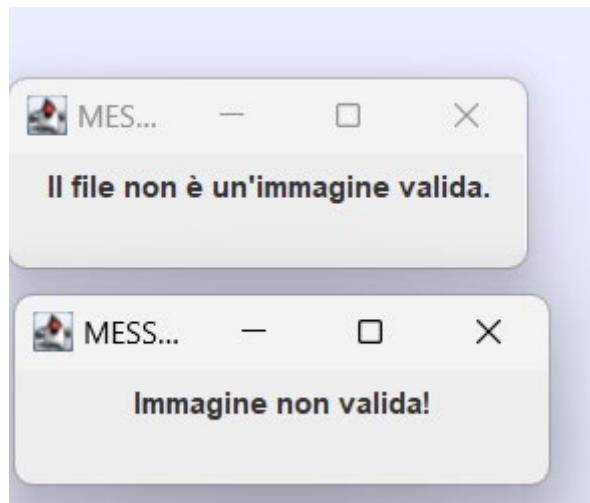
Password  
.....

Reinserisci password  
.....

Immagine del profilo (Max 5MB)  
Scegli il file fake-photo.jpg  
(Formati accettati: PNG,JPG,JPEG)

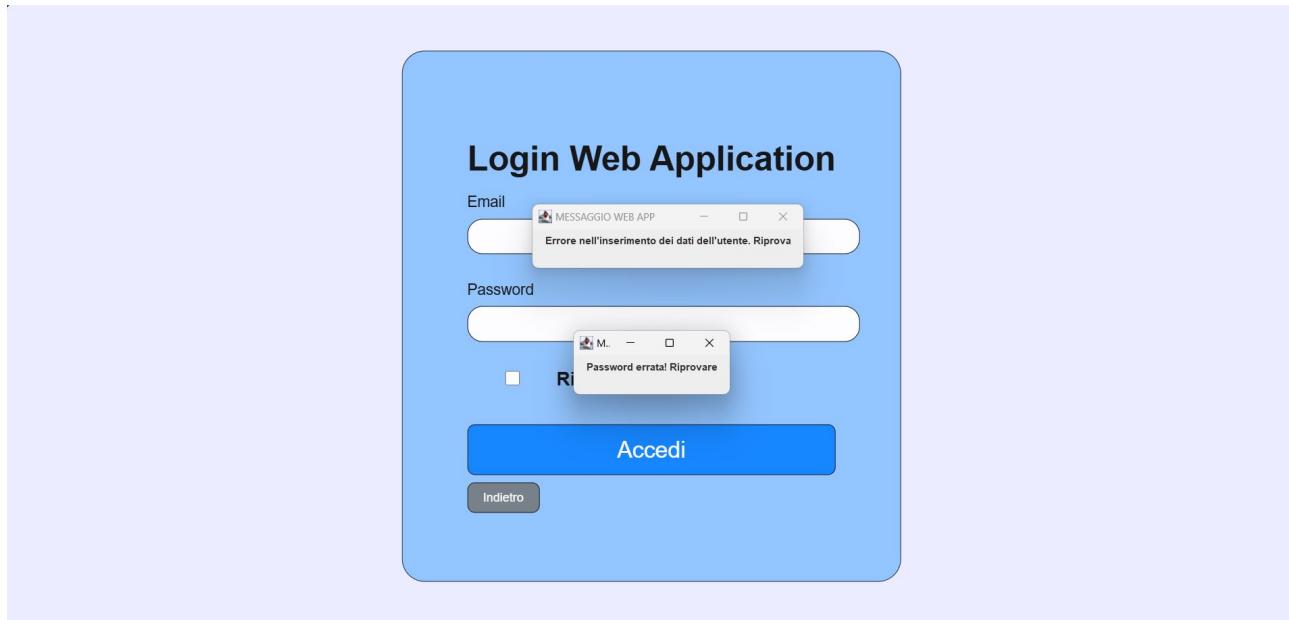
**Registrati**

**Indietro**



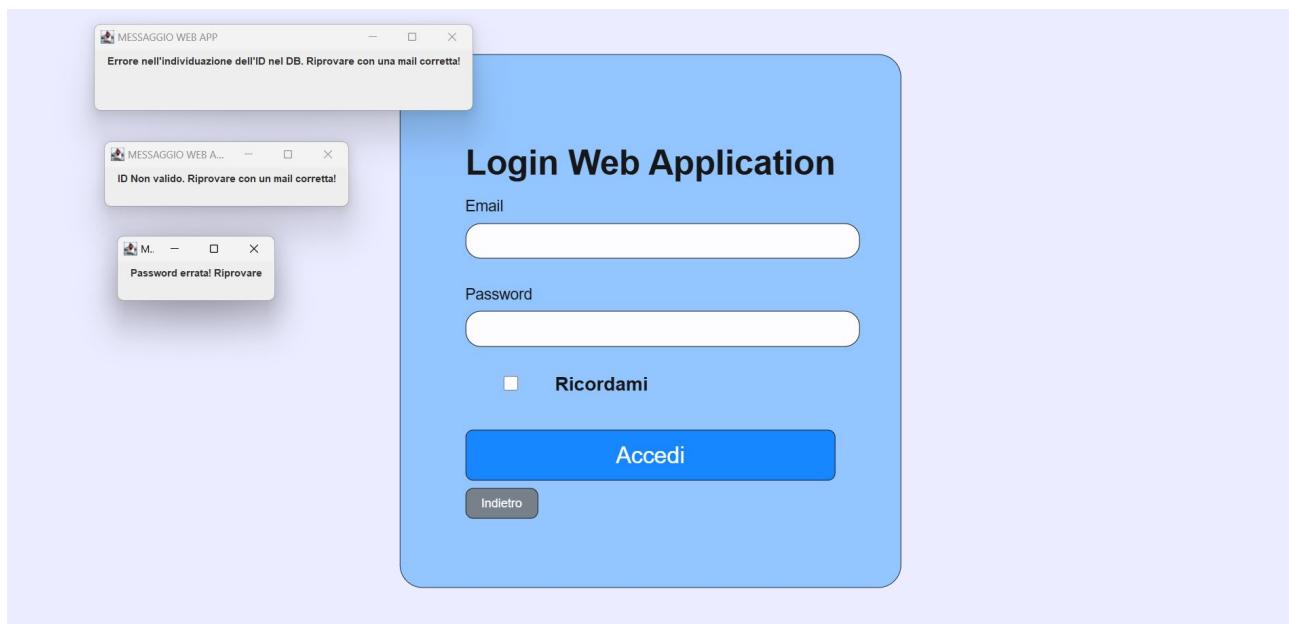
### Caso d'uso XV: Login con password errata

Descrizione	Pre-condizioni	Post-condizioni
L'utente effettua il login con una password errata	Nessuna	<ul style="list-style-type: none"> <li>Il sistema reindirizza l'utente sulla medesima pagina ripulendo tutti quanti i campi</li> <li>L'utente non accede alla pagina delle proposte</li> </ul>



## Caso d'uso XVI: Login con e-mail errata

Descrizione	Pre-condizioni	Post-condizioni
L'utente effettua il login con una mail errata	Nessuna	<ul style="list-style-type: none"> <li>Il sistema reindirizza l'utente sulla medesima pagina ripulendo tutti quanti i campi</li> <li>L'utente non accede alla pagina delle proposte</li> </ul>

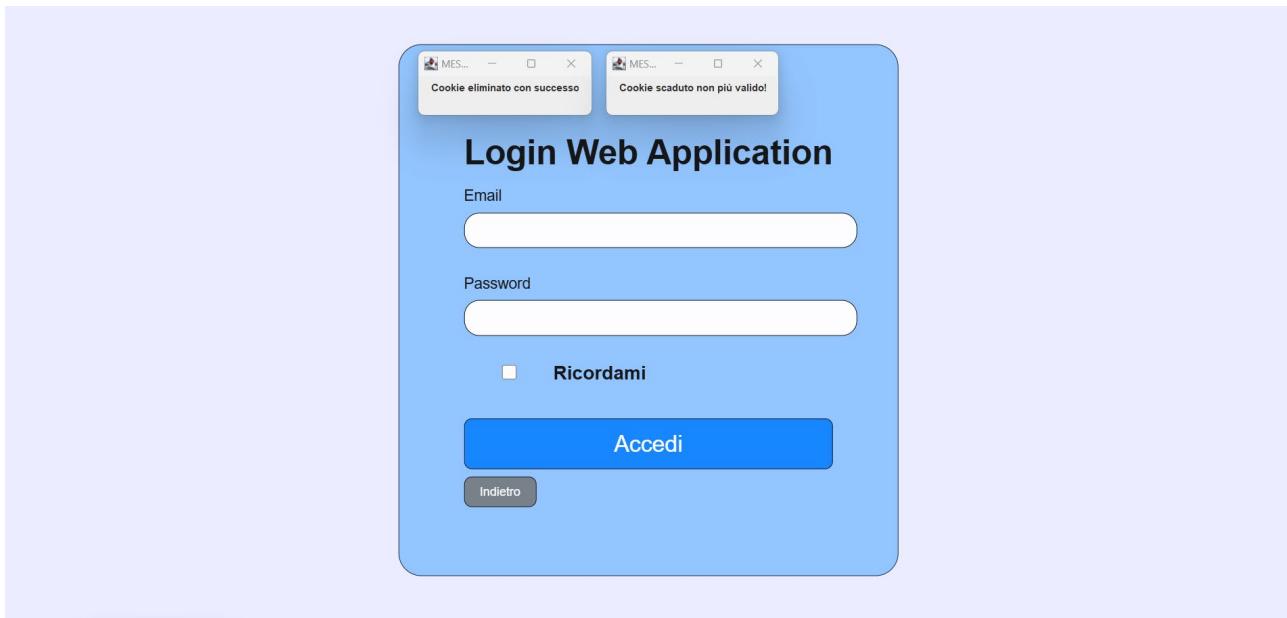


## Caso d'uso XVII: L'utente privo di cookie cerca di visitare la pagina protetta senza aver effettuato il login

Descrizione	Pre-condizioni	Post-condizioni
L'utente visita la pagina protetta	<ul style="list-style-type: none"> <li>• Utente non loggato</li> <li>• Sessione non valida a causa del timeout di 15 minuti</li> <li>• Cookie non registrati</li> </ul>	<ul style="list-style-type: none"> <li>• Il sistema reindirizza l'utente all'Homepage</li> </ul>

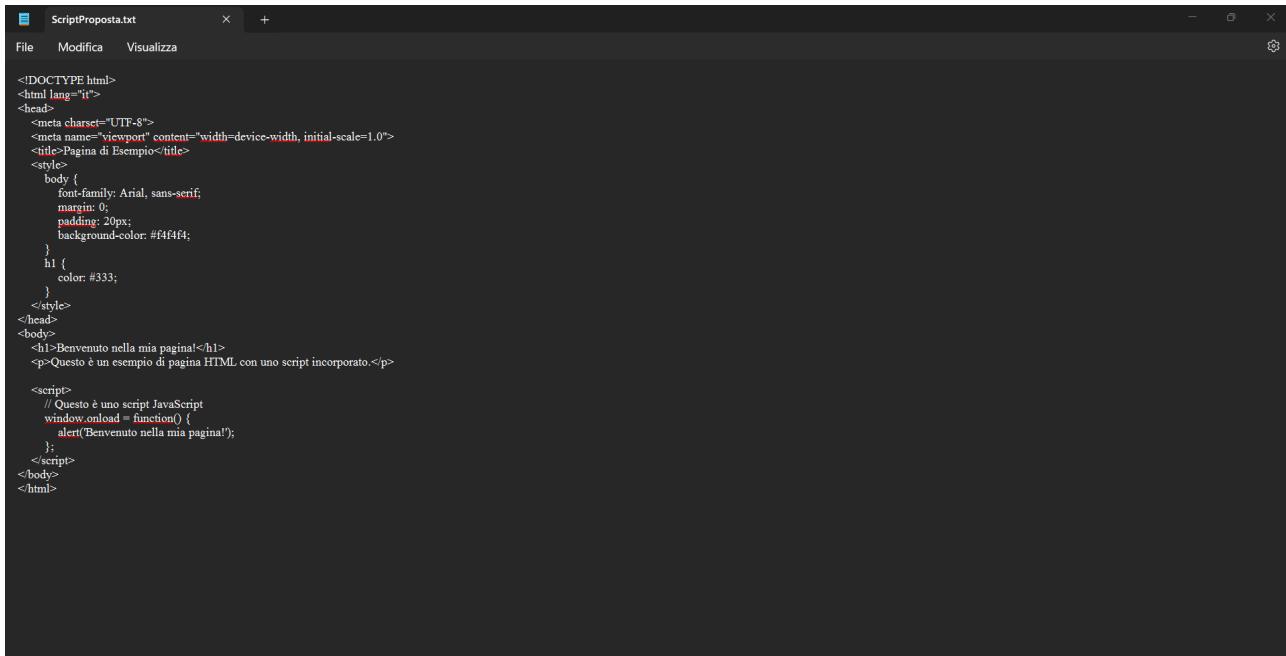
## Caso d'uso XVIII: L'utente con cookie registrato ma scaduto cerca di visitare la pagina protetta senza aver effettuato il login

Descrizione	Pre-condizioni	Post-condizioni
L'utente visita la pagina protetta	<ul style="list-style-type: none"> <li>• Utente non loggato</li> <li>• Sessione non valida a causa del timeout di 15 minuti</li> <li>• Cookie registrati scaduti</li> </ul>	<ul style="list-style-type: none"> <li>• Il sistema reindirizza l'utente all'Homepage</li> <li>• I cookie scaduti vengono eliminati</li> </ul>

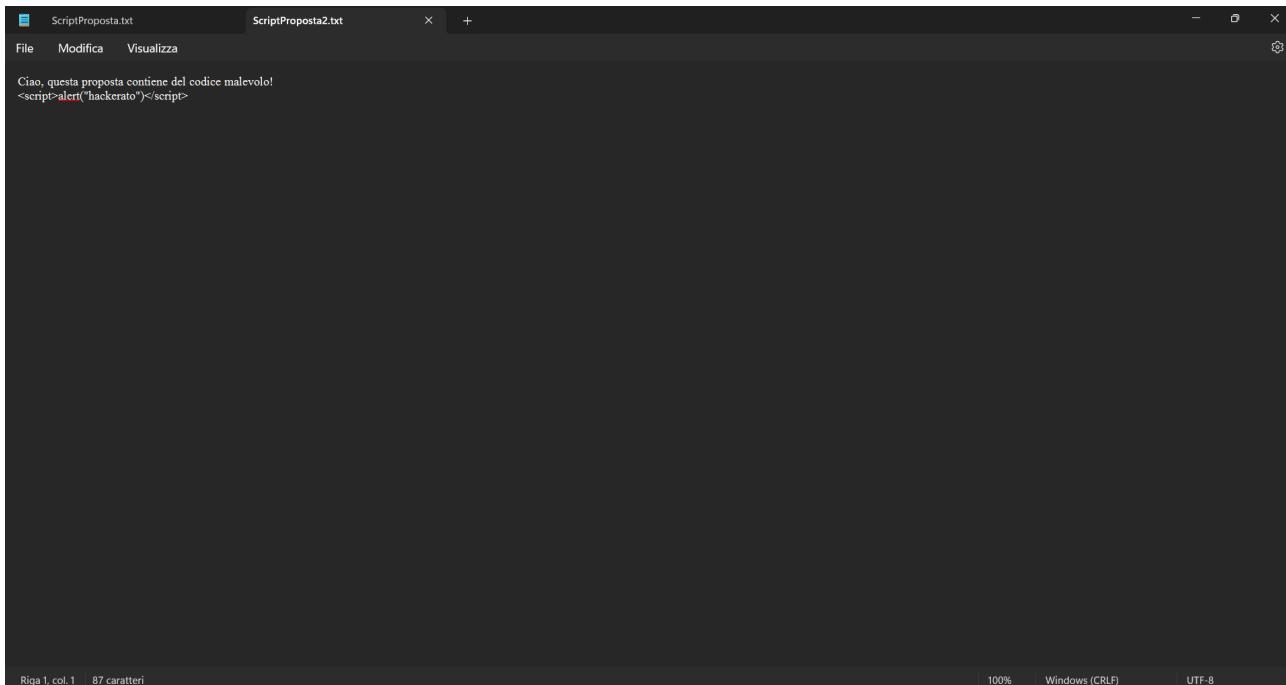


## Caso d'uso XIX: Caricamento della proposta con script incorporato

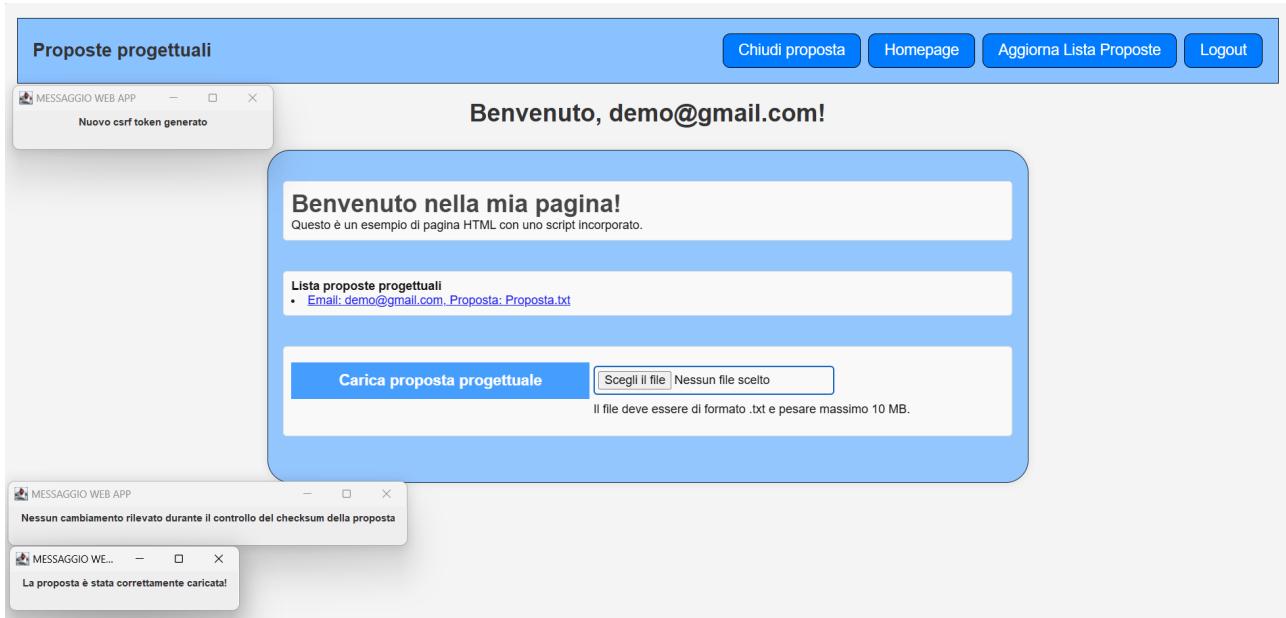
Descrizione	Pre-condizioni	Post-condizioni
L'utente effettua il caricamento di una proposta in formato txt con uno script malevolo incorporato	<ul style="list-style-type: none"> <li>• Utente loggato</li> <li>• Sessione valida</li> </ul>	<ul style="list-style-type: none"> <li>• Lo script viene rimosso</li> <li>• Il documento risulta caricato all'interno della Web App</li> </ul>



```
<!DOCTYPE html>
<html lang="it">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Pagina di Esempio</title>
<style>
body {
    font-family: Arial, sans-serif;
    margin: 0;
    padding: 20px;
    background-color: #f4f4f4;
}
h1 {
    color: #333;
}
</style>
</head>
<body>
<h1>Benvenuto nella mia pagina!</h1>
<p>Questo è un esempio di pagina HTML con uno script incorporato.</p>
<script>
// Questo è uno script JavaScript
window.onload = function() {
    alert("Benvenuto nella mia pagina!");
}
</script>
</body>
</html>
```



```
Ciao, questa proposta contiene del codice malevolo!
<script>alert("hackerato")</script>
```



Come si può notare all'interno del **Database**, lo script è stato correttamente rimosso:

A screenshot of a database editor titled "Edit Data for proposal\_content (BLOB)". The "Text" tab is selected, showing the following HTML code:

```

1  <!doctype html>
2  <html lang="it">
3  <head>
4  <meta charset="UTF-8">
5  <meta name="viewport" content="width=device-width, initial-scale=1.0">
6  <title>Pagina di Esempio</title>
7  <style>
8  body {
9      font-family: Arial, sans-serif;
10     margin: 0;
11     padding: 20px;
12     background-color: #f4f4f4;
13 }
14 h1 {
15     color: #333;
16 }
17 </style>
18 </head>
19 <body>
20 <h1>Benvenuto nella mia pagina!</h1>
21 <p>Questo è un esempio di pagina HTML con uno script incorporato.</p>
22 </body>
23 </html>

```

At the bottom, it says "Data Length: 570 bytes" and has "Save..." and "Load..." buttons.

## Caso d'uso XX: Caricamento di un file con estensione diversa da .txt

Descrizione	Pre-condizioni	Post-condizioni
L'utente effettua il caricamento di una proposta in un formato diverso da txt	<ul style="list-style-type: none"> <li>• Utente loggato</li> <li>• Sessione valida</li> </ul>	<ul style="list-style-type: none"> <li>• La proposta non viene caricata</li> </ul>

Proposte progettuali

Homepage Aggiorna Lista Proposte Logout

Benvenuto, demo@gmail.com!

**Lista proposte progettuali**

- Email: demo@gmail.com, Proposta: Proposta.txt
- Email: demo@gmail.com, Proposta: ScriptProposta.txt

**Carica proposta progettuale**

Scegli il file profile.jpg

Il file deve essere di formato .txt e pesare massimo 10 MB.

Carica Proposta

Proposte progettuali

Homepage Aggiorna Lista Proposte Logout

Benvenuto, demo@gmail.com!

**Lista proposte progettuali**

- Email: demo@gmail.com, Proposta: Proposta.txt
- Email: demo@gmail.com, Proposta: ScriptProposta.txt

**Carica proposta progettuale**

Scegli il file profile.jpg

Il file deve essere di formato .txt e pesare massimo 10 MB.

Carica Proposta

## Glossario

**Homepage:** Pagina principale accessibile a tutti gli utenti (Loggati e non loggati).

**Pagina delle proposte:** Pagina accessibile soltanto agli utenti loggati in cui è possibile visualizzare le proposte caricate da qualsiasi utente