Politecnico di Torino

Politecnico di Torino

Ingegneria Informatica

A.a. 2024/2025

Sessione di laurea Dicembre 2025

Analisi di una PWA e le sue funzionalità native con implementazione di autenticazione tramite fingerprint

Relatori:

Marco Torchiano

Salvatore Sabato

Candidati:

Michele Schiavone

Indice

Elenco delle figure

Capitolo 1

Introduzione

Il panorama delle applicazioni ha visto una crescita straordinaria negli ultimi anni, fornendo un numero eccessivo di applicazioni per ogni categoria. La varietà di piattaforme e software ha spinto i colossi informatici a stabilire standard, a causa della varietà di esigenze degli utenti. Il modo in cui le applicazioni moderne vengono sviluppate, distribuite e utilizzate è stato fortemente influenzato da aziende significative come Google, Apple e Microsoft.
Al tempo stesso, gli sviluppatori hanno dovuto ampliare le proprie conoscenze, imparando linguaggi e framework differenti per stare al passo e garantire prestazioni ottimali su ogni piattaforma. Grazie però a dei nuovi standard e strumenti messi a disposizione dalle grandi aziende, oggi è possibile sviluppare un'unica applicazione, sfruttando una sola tipologia di linguaggio, e garantire che questa sia automaticamente adatta a dispositivi e sistemi operativi differenti.

In questo contesto, troviamo le Progressive Web App, o PWA, che rappresentano una tecnologia moderna, che unisce il mondo delle web app a quello delle app native. Esse, infatti, combinano la leggerezza e la portabilità delle applicazioni web con la fluidità, le prestazioni e le funzionalità tipiche delle applicazioni installabili localmente, quali le notifiche push e l'accesso alle risorse hardware.
Ma, mentre la tecnologia avanza, siamo davvero sicuri che i nostri dati siano al sicuro? Negli ultimi anni, con il progresso della tecnologia, abbiamo ignorato gli effetti negativi. Abbiamo postato di tutto. Abbiamo concesso a terzi di accedere ai nostri dati, senza il nostro consenso. O, peggio, senza preoccuparci fino in fondo dell'uso che le aziende fanno dei nostri dati.
Contemporaneamente a questa evoluzione, la sicurezza informatica è diventata un elemento sempre più critico. L'aumento esponenziale delle applicazioni e dei servizi online ha, infatti, comportato una crescita altrettanto significativa dei rischi. Questi riguardano la gestione delle identità digitali e la protezione dei dati personali. I sistemi di autenticazione tradizionali, basati su password, hanno mostrato limiti

evidenti: le password sono deboli, vengono infatti riutilizzate o sono esposte a tecniche di phishing e brute force, e oggi rappresentano uno dei principali vettori di attacco.

Per rispondere a queste esigenze, il settore ha avviato una nuova transizione verso modelli di autenticazione più sicuri e trasparenti, come i nuovi standard passwordless e le autenticazioni biometriche, che forniscono ulteriori prove di autenticazione. Oggi, troviamo anche nuovi standard che si sono diffusi ampiamente: le passkey o i meccanismi biometrici integrati nei browser o nei sistemi operativi. Tuttavia, l'integrazione di queste tecnologie risulta complicata in architetture web complesse, come nel caso delle Progressive Web App. Infatti, si riscontrano ancora sfide significative, soprattutto in termini di gestione delle sessioni, compatibilità cross-platform e sicurezza dei token di accesso. La tesi affronta proprio il tema dell'autenticazione delle PWA, proponendo una soluzione moderna, scalabile, sicura e conforme agli standard di sicurezza più recenti.

L'obiettivo della tesi è quello di realizzare un sistema di autenticazione sicuro e scalabile per una Progressive Web App che sfrutta le WebAuthn API per accedere al sensore biometrico integrato, anche con il protocollo OAuth 2.0. Il lavoro prevede anche la realizzazione di test per valutare le prestazioni e la stabilità del sistema, attraverso test di carico, di picco e di stabilità, al fine di verificare la solidità del proxy sviluppato. Gli obiettivi specifici erano quelli di integrare il proxy con i provider esistenti e di migliorare la sicurezza e l'esperienza utente per rendere la PWA sviluppata il più facile possibile da usare.

Il progetto combina diverse tecnologie moderne: Angular per la parte frontend, Spring Boot per il backend e il proxy, sviluppato in analogia a OAuth2-Proxy e personalizzato per il caso specifico. L'uso delle WebAuthn API ha permesso di eliminare completamente l'uso delle password, garantendo un'autenticazione sicura grazie all'integrazione con il protocollo OAuth2.0 che, a sua volta, garantisce l'interoperabilità con provider esterni come Google.
Per semplicità, il lavoro è stato suddiviso in due fasi, seguendo un approccio incrementale. Nella prima fase, è stata condotta un'analisi sulle PWA già esistenti e sulle varie API disponibili, nonché sui metodi di autenticazione oggi sul mercato. Nella seconda parte, invece, è stata sviluppata una nuova PWA e il relativo sistema di autenticazione. Successivamente, sono stati condotti dei test utilizzando Apache JMeter.

In particolare, la tesi è organizzata come segue: la prima parte è suddivisa in tre capitoli. Il secondo capitolo introduce il contesto tecnologico, analizzando i concetti fondamentali delle Progressive Web App, mentre il terzo affronta il tema

della sicurezza e delle moderne tecniche di autenticazione. In questa parte si presta particolare attenzione alle soluzioni passwordless, necessarie per questo lavoro di tesi.
Nel quarto capitolo viene analizzato lo standard OAuth2 e il modo in cui viene integrato dai vari provider esterni, con un focus particolare sul meccanismo basato sui token, che risulterà necessario per l'integrazione con il proxy.
La seconda parte entra più nel dettaglio del lavoro ed è suddivisa in due capitoli. Il capitolo 5 descrive la Progressive Web App sviluppata, illustrando le interfacce e la tecnologia utilizzata e la logica di integrazione tra frontend, backend e proxy. Nel sesto capitolo viene invece descritto il lavoro di implementazione con il metodo di autenticazione tramite fingerprint, entrando più nel dettaglio del ruolo del proxy nell'autenticazione. In questo capitolo vengono anche presentati i test condotti per dimostrare l'efficienza del proxy anche con un carico elevato.
Infine, il settimo e ultimo capitolo contiene le conclusioni, le criticità riscontrate e le prospettive di sviluppo futuro.

Capitolo 2

Progressive Web App (PWA)

2.1 Definizione e caratteristiche principali

Grazie a nuove tecnologie e linguaggi di programmazione sofisticati come HTML5 e CSS3, il Web ha fatto passi da gigante negli ultimi anni, offrendo nuove funzionalità avanzate. Nuovi framework JavaScript come Angular e React hanno favorito lo sviluppo delle applicazioni a singola pagina (SPA), che offrono tempi di caricamento più rapidi e un'esperienza utente simile a quella delle applicazioni native, sviluppate appositamente per i dispositivi e scaricate quotidianamente dagli store ufficiali. Queste applicazioni, però, presentano qualche problema, come tempo di caricamento più lenti, o indisponibilità offline. [1].
Per sopperire a questa mancanza nasce il concetto di **Progressive Web App (PWA)**, con lo scopo di standardizzare lo sviluppo web moderno. Le PWA sono infatti applicazioni web che si comportano come app native, ma sono eseguite direttamente nel browser e sono disponibili su qualsiasi dispositivo, senza la necessità di essere sviluppate singolarmente per ogni sistema operativo, preferendo quindi vie più accessibili ed essere posti meno allo standard voluto dai proprietari dei dispositivi, con l'obiettivo di garantire un'esperienza utente uniforme, coerente per ogni utente, indipendentemente dal dispositivo utilizzato.
Come suggerisce il nome, le PWA adottano un approccio **progressivo**, adattandosi alle capacità del dispositivo e del browser su cui vengono eseguite. Ad esempio, una PWA che consente il caricamento di immagini può accedere alla fotocamera su un dispositivo mobile compatibile o limitarsi al caricamento di file su un desktop sprovvisto di fotocamera [2]. Questa adattabilità è resa possibile da un insieme di strategie e API e non dipende da nuovi framework o linguaggi.
Le PWA offrono poi un'esperienza veloce e stabile che non fa percepire la differenza

con un'applicazione nativa. Ciò è possibile grazie al caching e alla sincronizzazione in background, che garantiscono buone prestazioni anche in caso di connessioni Internet deboli o assenti. La differenza con un'app nativa è limitata inoltre dalla possibilità di inviare notifiche push, di essere installate sulla schermata iniziale o di sfruttare le funzionalità del dispositivo quando supportate.
Le caratteristiche chiave delle PWA includono:

- **Velocità**: grazie al caching, il contenuto può essere reso in pochi secondi e l'esperienza utente è immediata.
- **Affidabilità**: funzionano anche in condizioni di rete instabile o offline, mantenendo una buona fruibilità.
- **Coinvolgimento**: possono inviare notifiche push e installarsi come app, incentivando un'interazione continua con l'utente.
- **Adattabilità**: si adattano a qualsiasi dispositivo, indipendentemente dal sistema operativo, offrendo interfacce coerenti.
- **Indicizzabilità**: a differenza delle app native, le PWA sono accessibili via URL e indicizzabili dai motori di ricerca.
- **Installabilità**: possono essere aggiunte alla schermata iniziale senza passare per gli app store ufficiali.
- **Sicurezza**: l'utilizzo obbligatorio di HTTPS riduce il rischio di intercettazioni e garantisce una maggiore protezione dei dati utente.

Per concludere, il modello Progressive Web App unisce il mondo delle app native e del web offrendo una soluzione versatile, moderna che garantisce accessibilità, efficienza e sicurezza su scala globale [3].

2.2 Architettura di una PWA

Le Progressive Web App (PWA) si basano su un'architettura di framework e tecnologie innovative che mira a offrire una user experience analoga a quella delle applicazioni native.
Il cuore delle PWA è il cosiddetto App Shell, una struttura in HTML, CSS e JavaScript, progettato, infatti, per garantire un caricamento rapido e la visualizzazione di contenuti dinamici recuperati da API esterne. La separazione tra struttura (shell) e contenuto (dati) consente di ottenere prestazioni elevate anche in condizioni di connessione limitata o assente. [4]

La **teoria dell'App Shell** suggerisce di installare una UI minimale fin dal primo accesso, in modo che, nelle visite successive, venga recuperata dalla cache e

mostrata istantaneamente, mentre i dati vengono richiesti al server.
Questa separazione tra interfaccia e contenuto migliora l'esperienza utente, in quanto l'app risulta immediatamente utilizzabile. Elementi statici come barre degli strumenti, menu e splash screen fanno parte della shell e possono essere facilmente memorizzati nella cache. In alcuni casi, quando l'app è statica e priva di contenuti dinamici, l'intera applicazione può coincidere con la shell stessa.
Nella progettazione di una PWA, l'adozione del modello basato su App Shell implica che alcune risorse statiche locali, come la barra di navigazione e la home page, vengano caricate anche in assenza di connessione. I contenuti dinamici sono successivamente inseriti tramite JavaScript, con il supporto fondamentale del service worker, una componente chiave per il caching e il funzionamento offline. Il vantaggio principale di questa architettura è che, una volta memorizzata nella cache, la shell non deve essere ricaricata ad ogni visita: l'utente dispone sempre di una base HTML iniziale, anche in assenza di rete, e l'app viene poi aggiornata con i contenuti dinamici richiesti. Questo approccio consente di replicare in ambiente web le funzionalità tipiche delle app native. [5]
Il **service worker** rappresenta un elemento essenziale di ogni PWA ed è il componente che si occupa della memorizzazione nella cache dell'App Shell. La sua funzione è quella di lavorare in background su un thread separato rispetto al main thread del browser, senza accesso diretto al DOM, e di fornire funzionalità avanzate come il caricamento offline, la gestione delle notifiche push e la sincronizzazione in background. Al primo accesso dell'utente, la pagina web registra il service worker, il quale attiva un evento di installazione durante il quale viene memorizzata la shell. Questo worker controlla successivamente tutte le richieste di navigazione future, rendendo l'app sempre più reattiva [4].
Durante il ciclo di vita del service worker, una volta che il file JavaScript viene trovato e analizzato correttamente, si passa dallo stato di installazione a quello attivo. Se la registrazione fallisce, il processo viene interrotto. Quando il service worker è attivo, acquisisce il controllo dell'intera applicazione, consentendo operazioni come intercettazione delle richieste, caching delle risorse statiche e sincronizzazione. Il worker può entrare in uno stato di inattività quando non sono presenti eventi da gestire e, se questa condizione persiste, viene temporaneamente terminato dal browser, restando comunque pronto a riattivarsi al successivo evento.
Tuttavia, per motivi di sicurezza, i browser richiedono che la registrazione di un service worker avvenga in contesti sicuri, serviti cioè tramite HTTPS. Questo perché, utilizzando il service worker, è possibile dirottare le connessioni, creare e filtrare le risposte.[6]
Un service worker, nel suo ruolo funzionale, agisce come un "servitore" per la web app: esegue i compiti assegnati (le richieste) e lo fa anche in assenza di rete. Una volta che l'utente sarà nuovamente online, i dati immagazzinati torneranno disponibili. Il Worker collabora strettamente con le API del browser per il precaricamento

delle informazioni e la loro successiva distribuzione. Oltre alla cache delle risorse statiche, gestisce anche le richieste di rete, consentendo di amministrare più cache contemporaneamente. Ciò consente all'app di utilizzare meno dati e di funzionare in modo efficace sui dispositivi mobili nelle aree con scarsa connettività a Internet.[4]
Il ruolo dei service worker è stato anche analizzato in termini di impatto energetico: uno studio condotto da Malavolta ha dimostrato che il loro utilizzo non incide in maniera significativa sul consumo energetico, a prescindere dalla qualità della connessione o dal dispositivo utilizzato. Inoltre, la sincronizzazione in background consente alle PWA di attendere condizioni di rete ottimali per trasmettere le informazioni. Un esempio pratico è l'invio di un modulo compilato offline: la richiesta viene temporaneamente memorizzata dal service worker e inviata non appena disponibile una connessione.
Un altro elemento chiave nell'architettura di una PWA è il file manifest, un documento in formato JSON che consente agli sviluppatori di definire determinate caratteristiche dell'applicazione, come il nome visualizzato, l'icona, la schermata iniziale e altri parametri. Questo file consente di adattare il comportamento della PWA in modo che l'esperienza dell'utente sia simile a quella delle applicazioni native, rendendo la web app installabile e integrabile nel sistema operativo del dispositivo. [7]

2.3 Funzionalità native disponibili

Le Progressive Web App (PWA) si configurano come una soluzione ibrida che coniuga i vantaggi delle applicazioni web tradizionali e delle applicazioni mobili native. In linea con il principio del "best of both", esse permettono all'utente di usufruire di un'applicazione web tramite un browser, offrendo al contempo la possibilità di installarla sulla schermata iniziale del proprio dispositivo. Tra i principali promotori di questa tecnologia si colloca il gruppo Google Web Fundamentals, che ne ha sostenuto la diffusione e l'evoluzione [6].

Le PWA si distinguono dalle comuni web app per l'integrazione di comportamenti tipici delle applicazioni native. Tra questi, si annoverano il supporto al funzionamento offline, l'installabilità sul dispositivo e la capacità di inviare notifiche push. Ciò è reso possibile grazie a un insieme sinergico di tecnologie, concetti e API web, che lavorano insieme per offrire un'esperienza d'uso assimilabile a quella delle applicazioni mobili, senza però imporre i vincoli tipici di queste ultime.

Una delle funzionalità centrali offerte dalle PWA è la possibilità di operare in modalità offline. Tale capacità è abilitata dai **service worker**, uno degli elementi chiave dell'architettura delle applicazioni progressive. In effetti, le limitazioni strutturali delle app web tradizionali, come la dipendenza continua da una connessione di rete, la scarsa interattività e le prestazioni limitate, hanno sollevato nel

tempo la necessità di soluzioni più performanti. A differenza delle app native, le quali garantiscono una maggiore reattività e accesso diretto alle funzionalità del dispositivo, ma a costi e tempi di sviluppo più elevati, le PWA si presentano come un compromesso efficace. Esse offrono elevate prestazioni unite a una maggiore scalabilità, consentendo di evitare lo sviluppo e la manutenzione di basi di codice distinte per ciascuna piattaforma. [2]

Il contributo del **service worker** si estende oltre la semplice disponibilità offline. Esso consente, infatti, di pre-caricare risorse nella cache e di servire contenuti anche in assenza di rete. Inoltre, grazie alla **background sync**, è possibile memorizzare localmente le richieste effettuate senza connessione, per poi eseguirle non appena la rete viene ristabilita. Le richieste, una volta generate, vengono salvate attraverso l'uso dell'API **IndexedDB**, in modo tale che anche la chiusura dell'applicazione non impedisca il successivo completamento delle operazioni. Questo sistema garantisce una sincronizzazione sicura e affidabile tra i dati offline e quelli presenti sul server, migliorando l'integrità e la coerenza delle informazioni trattate [7].

Poi c'è un altro vantaggio: il carico di lavoro sui server è inferiore poiché molte risorse vengono conservate localmente. Le interazioni successive con le app riducono significativamente la necessità di accessi continui alla rete, limitando l'utilizzo della larghezza di banda e riducendo la latenza. Le operazioni offline, oltre a garantire una maggiore fluidità dell'esperienza utente, consentono anche un uso più efficiente del contenuto dei dati mobili.

A rafforzare ulteriormente il coinvolgimento dell'utente contribuiscono le notifiche push, considerate una delle funzionalità principali delle PWA. Questi avvisi, visibili sul dispositivo, possono essere inviati anche quando l'app è chiusa o non in uso. Il loro obiettivo è mantenere alta l'attenzione e la partecipazione dell'utente fornendo aggiornamenti puntuali e pertinenti. Per funzionare, questo meccanismo richiede l'integrazione della Push API che, lavorando in stretta collaborazione con il *service worker*, consente la ricezione e la visualizzazione delle notifiche in modo asincrono e persistente [4].

2.4 Accesso allo storage e fotocamera

Tra le funzionalità oggi disponibili per le Progressive Web App (PWA) che si avvicinano in modo significativo a quelle delle applicazioni native, rivestono particolare importanza l'accesso allo storage e alla fotocamera. Le PWA possono interagire con la fotocamera, il microfono e lo spazio di archiviazione locale sfruttando le normali API Web, nel rispetto di vincoli di sicurezza come l'utilizzo del protocollo HTTPS e l'ottenimento esplicito del consenso da parte dell'utente [8].

2.4.1 Accesso ai flussi multimediali

Le PWA possono usare la fotocamera e i flussi multimediali grazie a un'API principale che fa parte dell'ecosistema WebRTC: la Media Capture and Streams API. Grazie all'interfaccia `navigator.mediaDevices.getUserMedia()` è possibile inizialmente chiedere i consensi al sistema, per permettere all'applicazione di accedere alla webcam e al microfono, accettando come parametro un oggetto che definisce i vincoli desiderati (per esempio *{video: true, audio: true}*). Se l'utente dà il consenso e il dispositivo ha l'hardware necessario, viene restituito un oggetto *MediaStream* che contiene una o più tracce video e/o audio. Altrimenti, si verifica un errore come *NotAllowedError* (permesso negato) o *NotFoundError* (dispositivo non disponibile) [9]. Inoltre, la funzione *getUserMedia* consente di personalizzare i vincoli, specificando ad esempio la risoluzione, il frame rate o la scelta tra la fotocamera anteriore e quella posteriore.

Una chiamata come

```
1    video: { facingMode: { exact: "environment" } }
```

permette di selezionare esplicitamente la fotocamera posteriore di un dispositivo mobile, ma se il vincolo non può essere soddisfatto, la promessa viene rifiutata con *NotFoundError*. È fondamentale notare che questa API è disponibile solo in contesti sicuri, cioè su siti serviti tramite HTTPS o in ambiente localhost durante lo sviluppo: senza queste condizioni, l'oggetto *navigator.mediaDevices* risulta indefinito e qualsiasi tentativo di utilizzo fallisce automaticamente. Ogni invocazione dell'API comporta l'apertura di un dialogo di autorizzazione da parte del browser, il quale richiede esplicitamente il consenso dell'utente prima di concedere l'accesso ai dispositivi. [10]

Il flusso multimediale restituito da *getUserMedia* può essere gestito in tempo reale grazie alla **Streams API**, che consente alla PWA di elaborare i dati in ingresso dinamicamente. I frame video possono essere visualizzati, filtrati, registrati o trasmessi a un server, offrendo così un'ampia gamma di funzionalità multimediali direttamente nel browser.
L'adozione di questa API è oggi largamente diffusa: è supportata dalla quasi totalità dei browser moderni, inclusi Chrome, Firefox, Edge e Safari (dalla versione 11 in poi su desktop e mobile). Da Chrome 47 in avanti, inoltre, l'uso di *getUserMedia* è esplicitamente disabilitato in contesti non sicuri, rafforzando ulteriormente le garanzie di sicurezza .

2.4.2 Accesso allo storage

In relazione allo storage locale, le PWA possono salvare e gestire dati lato client attraverso diverse API, ognuna pensata per specifici scenari. Le più comuni sono le

Web Storage API, ovvero *localStorage* e *sessionStorage*.
Queste API consentono di memorizzare coppie chiave/valore, come stringhe, in modo più semplice e diretto rispetto ai cookie. Il localStorage può essere utile per conservare dati persistenti che saranno disponibili anche quando l'utente ha chiuso il browser e fornisce un area di archiviazione condivisa per ogni origine (protocollo, host e porta), spesso impiegato per memorizzare preferenze, configurazioni e dati in cache.
Il sessionStorage, invece, conserva i dati solo per la durata della sessione della singola tab, risultando separato per ogni finestra o scheda del browser. Una volta chiuso il tab, tutte le informazioni vengono automaticamente eliminate. Entrambe le API operano in modo sincrono: ciò significa che le operazioni di lettura o scrittura bloccano l'esecuzione del codice finché non si concludono, aspetto che le rende meno adatte per gestire dataset di grandi dimensioni.

Con volumi di dati più consistenti, è preferibile usare **IndexedDB**, un database locale asincrono basato su oggetti che consente di memorizzare strutture dati complesse, incluse stringhe, numeri, oggetti JSON e file binari (blob). Con IndexedDB si può migliorare il tempo di caricamento salvando lo stato dell'applicazione o memorizzando i contenuti scritti dall'utente in attesa di sincronizzazione con un server remoto. Ma se non utilizzato correttamente, si potrebbero verificare problemi di mantenibilità, soprattutto senza una gestione attenta delle versioni del database o di meccanismi di pulizia.

Un'altra API correlata è la **Cache API**, legata al funzionamento dei Service Worker.
Essa consente di catturare le richieste di rete e di fornire le risorse da una cache locale, assicurando così che l'applicazione funzioni anche senza connessione. In questo modo, si ottimizzano sia le prestazioni che la resilienza, poiché le risorse (come HTML, JavaScript, CSS e immagini) vengono prelevate direttamente dal dispositivo anziché dalla rete. La persistenza delle risorse è garantita fino alla loro eliminazione esplicita da parte dell'applicazione, e la gestione avviene attraverso metodi asincroni, come `cache.put()` e `cache.addAll()`. Anche in questo caso, lo spazio disponibile dipende dalle politiche del browser, ma può arrivare a centinaia di megabyte, rendendo la Cache API particolarmente adatta per applicazioni di medie o grandi dimensioni. Tutti i browser moderni che supportano i Service Worker includono anche la Cache API, inclusi Safari (a partire dalla versione 11 su desktop e dalla versione 14 su iOS).

Infine, per scenari più avanzati in cui è necessario leggere e scrivere file sul file system locale, le PWA possono sfruttare la **File System Access API**, disponibile in due modalità principali.
La prima, tramite l'utilizzo di file picker, permette all'utente di scegliere file locali attraverso finestre di dialogo gestite dal sistema operativo (come `window.showOpenFilePicker()`).
Una volta selezionati, i file possono essere letti con getFile() o modificati tramite

`createWritable()`. Per motivi di sicurezza, è possibile modificare i file solo dopo che l'utente ha fornito l'autorizzazione, rendendo questa modalità adatta per applicazioni più complesse, come editor, IDE o strumenti di elaborazione multimediale, ma attualmente è supportata solo dai browser basati su Chromium, quindi non disponibile su Safari né su Firefox.
La seconda modalità è denominata Origin Private File System (OPFS) e fornisce uno spazio di archiviazione isolato, non visibile all'utente, pensato per dati di grandi dimensioni da utilizzare offline. Si accede a questo file system tramite `navigator.storage.getDirectory()`, e il contenuto viene automaticamente eliminato quando l'utente cancella i dati del sito.[11]

2.5 Plugin e API browser avanzate

Oggigiorno, i browser moderni non possono essere più considerati semplici strumenti per visualizzare delle pagine web, ma come delle vere e proprie piattaforme applicative. Infatti oggi possono eseguire del codice, interagire con l'hardware o addirittura comunicare con dispositivi esterni. Se oggi tutto questo è possibile, è sia grazie all'introduzione dei plugin (oggi in gran parte obsoleti) e sia allo sviluppo si API avanzate, in certi casi messe a disposizione del browser stesso. In questo contesto, è fondamentale comprendere quali sono le funzionalità avanzate fornite dai browser.

2.5.1 Notifiche: Notifications API e Push API

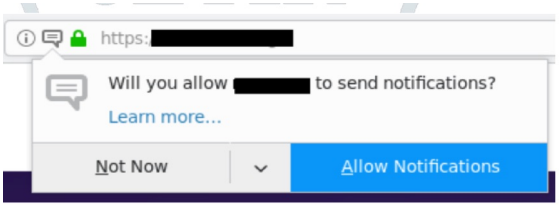

Figura 2.1: Richiesta di autorizzazione per notifiche da parte di una PWA

Un'API particolarmente interessante nel panorama delle API avanzate è quella che si occupa delle notifiche, rappresentata dal sistema **Web Push**. Questa si articola in due API: **Notifications API** e textbfPush API, che consentono alle applicazioni web di interagire con l'utente, anche al di fuori del normale ciclo di utilizzo attivo dell'applicazione.

Con la Notifications API, i siti web hanno la possibilità di inviare messaggi istantanei agli utenti mentre stanno navigando, facendo comparire le classiche notifiche pop-up. Ovviamente, sempre per motivi di sicurezza, anche l'utilizzo di

questa API richiede il consenso esplicito dell'utente.
Le notifiche vengono gestite tramite l'integrazione con i service worker, che permettono la visualizzazione anche in condizioni di attività ridotta della pagina.

La Push API, invece, estende questa funzionalità consentendo la consegna asincrona di notifiche anche in assenza di una sessione attiva del browser o quando la relativa scheda è chiusa. Nonostante il potenziale elevato di questa tecnologia, va segnalato che non tutti i browser supportano pienamente la Push API, in particolare all'interno dell'ecosistema Apple e su dispositivi iOS, dove il supporto risulta più limitato o soggetto a restrizioni.

Per garantire il corretto funzionamento del sistema di notifica, è necessario l'intervento di un servizio push integrato nel browser. Ogni browser moderno implementa un proprio servizio push, responsabile della gestione, dell'instradamento e della consegna dei messaggi verso il client corretto. Il processo completo di notifica consiste in tre fasi principali: innanzitutto, è necessario abilitare le notifiche sul dispositivo dell'utente; successivamente, l'applicazione verifica l'esistenza di subscriptions attive tramite i service worker; infine, qualora queste subscriptions non fossero presenti, ne viene generata una nuova, che viene registrata sul server back-end dell'applicazione.

In questa fase vengono anche create delle chiavi particolari, dette *VAPID (Voluntary Application Server Identification)*, generate localmente e che identificano l'applicazione e garantiscono una comunicazione sicura (protetta da crittografia). I messaggi vengono poi inoltrati dal server al servizio push del browser che li mette in coda, aspettando la riconnessione del dispositivo. Quando la rete è disponibile, il messaggio scatena un push event che 'risveglia' il service worker per mostrare l'avviso e che può essere riattivato da un click dell'utente sulla notifica, chiudendo il ciclo di vita del messaggio.

Nel complesso, la combinazione di Push API e Notifications API consente di realizzare interazioni persistenti e reattive tra l'applicazione web e l'utente, contribuendo a ridurre le distanze tra web app e applicazioni native. Tuttavia, è essenziale considerare attentamente le implicazioni in termini di sicurezza e privacy, in quanto l'uso improprio o non autorizzato di tali API potrebbe trasformarsi in un potenziale vettore d'attacco o in uno strumento invasivo nei confronti dell'utente [5].

2.5.2 Web Bluetooth API

I browser moderni possono scoprire, connettersi e comunicare con i dispositivi Bluetooth Low Energy (BLE) vicini grazie alla **Web Bluetooth API**, consentendo di stabilire delle connessioni con dispositivi Bluetooth per poter accedere ai dati in lettura o scrittura a specifiche caratteristiche offerte dal dispositivo. Infatti, in questo modo, si possono raccogliere dati necessari all'app, che possono variare da

dati biometrici, provenienti da sensori di frequenza cardiaca, a informazioni sul livello della batteria.

Entrando più nel dettaglio, anche l'utilizzo di questa API è vincolato a specifici requisiti di sicurezza: può essere richiamata solo da pagine servite tramite HTTPS e richiede un'azione esplicita dell'utente, come ad esempio un click per avviare la ricerca dei dispositivi. Dopo aver stabilito la connessione, si può avere interazione con il dispositivo BLE, ma solo all'interno del contesto del browser.

Nonostante il suo potenziale, la Web Bluetooth API è attualmente ancora considerata sperimentale. Esiste una specifica in fase di definizione da parte del Web Bluetooth Community Group, ma essa non è ancora stata ufficialmente approvata come W3C Recommendation. Di conseguenza, il supporto tra i browser risulta molto disomogeneo. L'API è disponibile principalmente sui browser basati su Chromium (Chrome, Microsoft Edge e Opera per desktop), mentre è assente su Firefox e Safari. In ambito mobile, la compatibilità è garantita su Chrome per Android (nelle versioni più recenti), Brave, Samsung Internet e Opera Mobile, ma rimane non disponibile su iOS, né tramite Safari né attraverso Chrome per iOS, a causa delle restrizioni imposte dal sistema operativo.

moltiplicando, soprattutto nel contesto dell'Internet of Things (IoT) e dei dispositivi wearable. Tra gli scenari concreti si considerano, ad esempio, il controllo remoto di periferiche come LED o motori collegati a un Raspberry Pi, oppure la raccolta di dati da sensori indossabili. Tali funzionalità, un tempo prerogativa esclusiva delle applicazioni native, stanno diventando sempre più accessibili anche all'interno del mondo web, pur permanendo limitazioni legate alla compatibilità, alla sicurezza e alla maturità della specifica [12].

2.5.3 Web USB API

La **Web USB API**, sebbene in fase di sperimentazione, rappresenta un'evoluzione significativa, consentendo alle pagine web di interagire direttamente con le periferiche USB connesse all'host. Ciò elimina la necessità di driver o software supplementari per l'interfacciamento con hardware non supportato nativamente.

L'attivazione richiede l'invocazione di navigator.usb.requestDevice(), dove vengono specificati criteri di selezione (come *Vendor ID* e *Product ID*). A seguito del consenso dell'utente, l'API restituisce un oggetto *USBDevice* che facilita una comunicazione bidirezionale e un trasferimento dati paragonabile a quello nativo.

Come già accennato, anche la Web USB API è attualmente considerata sperimentale dal punto di vista della standardizzazione, infatti non ha ancora raggiunto lo status di raccomandazione ufficiale del W3C ed è classificata come tecnologia non stabile anche dal Mozilla Developer Network (MDN).

Per quanto riguarda la disponibilità sui browser, la compatibilità è limitata ai browser basati su Chromium. Al contrario, non è disponibile né su Firefox, né su

Safari nella versione desktop. Anche sul fronte mobile, la situazione resta simile: Chrome per Android include il supporto alla Web USB API, ma non è presente su iOS, né attraverso Safari né tramite Chrome o altri browser, a eccezione di alcuni browser cinesi che hanno introdotto il supporto autonomamente.

L'utilizzo della Web USB API trova particolare applicazione in scenari di nicchia ma altamente specializzati, come l'interfacciamento con microcontrollori, il controllo di stampanti 3D, o la gestione di apparecchiature da laboratorio direttamente da una pagina web. Questi casi d'uso, che fino a poco tempo fa richiedevano ambienti desktop dedicati o software proprietari, stanno diventando sempre più accessibili tramite il browser, con un evidente vantaggio in termini di portabilità, disponibilità cross-platform e riduzione della complessità di distribuzione [13].

2.5.4 Web NFC API

Con questa API, le applicazioni web possono interagire con i **tag NFC passivi**. È un enorme aiuto in ambito mobile per operazioni veloci di lettura/scrittura, utilissime in campi come la logistica, il turismo digitale e l'interazione museale, offrendo un'esperienza utente fluida e integrata nel browser, senza la necessità di installare app dedicate.
La **Web NFC API** permette di leggere e scrivere dati su tag NFC non appena il dispositivo mobile si avvicina al tag (entro i 10 cm). Attraverso lo scambio, che è basato su uno standard per la serializzazione dei messaggi NFC chiamato **NDEF (NFC Data Exchange Format)**, l'applicazione Web può interagire con il tag per acquisire o sovrascrivere stringhe testuali, URL e payload compatibili.
Il meccanismo dell'API è molto semplice: un'app web (PWA) deve prima di tutto istanziare un oggetto `NDEFReader`. Successivamente, chiama `scan()` per avviare la ricerca dei tag e registra un event handler (`onreading`) che si occupa di elaborare i messaggi non appena viene rilevato un segnale NFC. Il modello di interazione asincrono rende l'API ideale per una vasta gamma di applicazioni che richiedono risposte immediate.
Tuttavia, la Web NFC API è ancora considerata sperimentale. Attualmente esiste un **Working Draft del W3C** dedicato alla standardizzazione di questa API, ma non è ancora una raccomandazione ufficiale. MDN la classifica come una tecnologia non baseline, ovvero non supportata universalmente nei principali ambienti di esecuzione, e sottolinea il carattere limitato e sperimentale della sua implementazione. Il sito caniuse.com conferma che, ad oggi, l'unico browser con supporto attivo alla Web NFC API è Google Chrome per Android. Nessun altro browser, inclusi Chrome desktop, Microsoft Edge, Firefox e Safari, implementa attualmente questa API.
La Web NFC API risulta particolarmente utile per applicazioni mobile in cui è richiesto lo scambio veloce di dati semplici tramite un semplice "tap". Alcuni casi

d'uso concreti includono: interazioni in musei e gallerie d'arte, dove l'utente può toccare un tag NFC per ricevere automaticamente informazioni su un'opera esposta; sistemi di gestione dell'inventario, dove i tag applicati a oggetti o contenitori vengono letti/scritti per aggiornare informazioni logistiche; oppure check-in digitali in fiere, congressi ed eventi, dove i badge NFC vengono utilizzati per l'identificazione rapida dei partecipanti essaggi quando viene rilevata una presenza NFC [14].

2.5.5 Payment Request API

Pensata per migliorare i moduli web tradizionali, questa tecnologia consente alle web app di usare i metodi di pagamento nativi del dispositivo, il che significa un netto miglioramento della velocità del checkout, anche in termini di sicurezza e usabilità. Avendo ricevuto forte impulso da entità come Google e Apple, si distingue come una delle Web API avanzate ad aver raggiunto la standardizzazione W3C ed è quindi pronta per l'impiego in contesti di produzione.
La Payment Request API permette alle app web di creare una richiesta di pagamento attraverso l'oggetto `PaymentRequest` che elenca i sistemi accettati (es. carte di credito o wallet digitali), i costi (totale, tasse, ecc.) e, a volte, i dati richiesti all'utente (come l'indirizzo). Una volta impostata, il comando `show()` fa apparire un box di dialogo nativo che mostra all'utente le sue opzioni di pagamento salvate. Così, la transazione viene completata in modo rapido e immediato.
Una volta completato il pagamento, l'API restituisce un oggetto (`PaymentResponse`) che contiene tutti i dati scelti dall'utente: dalla carta di credito all'indirizzo di spedizione, fino a ogni dettaglio necessario per concludere l'ordine, eliminando la necessità di compilare lunghi moduli e riducendo il rischio di errori. Questo sistema permette inoltre di ridurre il numero di utenti che abbandonano il carrello. Per questo motivo, è ormai usata ovunque: e-commerce, siti di prenotazione e servizi online.
A differenza di molte altre API avanzate, la Payment Request API non è sperimentale. È uno standard ufficiale W3C già in stato di Raccomandazione. Viene infatti ampiamente adottata da browser principali: è supportata in Google Chrome (desktop e Android), in Microsoft Edge, in Opera, e anche in Samsung Internet. Apple ne ha implementato il supporto in Safari desktop e iOS a partire dalle versioni 12.1/12.2. Safari 11.1–12 presenta un supporto parziale, mentre Firefox mantiene l'API disabilitata per impostazione predefinita, rendendola di fatto non utilizzabile in ambienti standard.
Nonostante questa ampia diffusione, **MDN** segnala la Payment Request API come non baseline, ossia non garantita universalmente su tutti i browser e piattaforme, ma non la classifica come sperimentale. Al contrario, viene considerata matura e pronta per la produzione, e il suo utilizzo è incoraggiato laddove si voglia integrare un flusso di pagamento più fluido e coerente con l'esperienza utente del dispositivo.

Tra i casi d'uso reali, la Payment Request API viene utilizzata per mostrare elenchi di carte di pagamento salvate, integrare wallet digitali come Google Pay o Apple Pay (quest'ultimo tramite WebKit o wrapper), oppure offrire opzioni di pagamento semplificate tramite PayPal, carte di credito/debito o app bancarie. Il vantaggio principale risiede nell'unificazione del flusso di pagamento, che può avvenire senza moduli HTML personalizzati, garantendo maggiore coerenza, sicurezza e semplicità per l'utente [15].

2.5.6 Web Share API

La **Web Share API** permette ai siti web di usare il pannello di condivisione nativo del telefono (o del PC). In questo modo, l'utente può condividere contenuti come link, testi o file a qualsiasi app installata (come WhatsApp, email o altre app). Questa API segna un'importante evoluzione, fornendo alle PWA un'integrazione di sistema comparabile a quella delle applicazioni native, in particolare in contesti mobili.
Per condividere un contenuto, si chiama il metodo `navigator.share(shareData)`. L'oggetto `shareData` definisce cosa inviare (titolo, testo, URL e, se supportato, anche file). Per ragioni di sicurezza è fondamentale che questo avvenga solo dopo un'interazione diretta dell'utente.
Un tipico scenario d'uso è quello di un sito di notizie che, alla pressione del pulsante "Condividi", chiama `navigator.share(title, text, url)`, consentendo all'utente di condividere l'articolo su WhatsApp, Telegram, Twitter, o qualsiasi altra app disponibile sul dispositivo. Questo approccio rende l'esperienza utente molto più fluida e coerente con quella delle applicazioni native.
Dal punto di vista della maturità tecnologica, la Web Share API non è sperimentale: è stabile, ben documentata su MDN e ampiamente implementata nei principali browser moderni. Tuttavia, presenta ancora alcune differenze di compatibilità, in particolare su desktop. È supportata in tutti i browser Chromium-based (Chrome, Edge, Opera, Brave, ecc.) sia su desktop che mobile, e in Safari su macOS e iOS. Anche browser mobili come Samsung Internet e Opera Mobile offrono il supporto. L'unica eccezione significativa è Firefox desktop, che non implementa l'API (né prevede di farlo a breve termine), limitando quindi l'utilizzo della funzionalità in ambienti desktop non-Chromium.
MDN la considera una funzionalità affidabile e pronta per l'uso in produzione, purché venga effettuato un controllo di compatibilità (`if (navigator.share) ...`) per gestire i casi in cui l'API non sia disponibile. Non è indicata come "baseline", ma si tratta comunque di una tecnologia ormai consolidata e ampiamente utilizzata, soprattutto nei contesti mobile[16].

La **Screen Wake Lock API**, parte del più ampio gruppo delle Wake Lock API, consente a una pagina web attiva di impedire lo spegnimento o l'oscuramento

automatico dello schermo del dispositivo. Questa funzionalità si rivela particolarmente utile in tutti quei contesti applicativi in cui l'utente deve poter consultare continuamente delle informazioni a schermo senza dover interagire attivamente, come ad esempio durante l'uso di un navigatore, la lettura di un ebook, la visualizzazione di presentazioni, l'uso di timer o strumenti di scansione in ambito logistico o di accesso.
L'uso dell'API si basa sulla chiamata al metodo `navigator.wakeLock.request ('screen')`, che consente di acquisire un wake lock di tipo screen. Una volta ottenuto il permesso, il sistema mantiene lo schermo attivo e impedisce che si attivi il risparmio energetico. La funzione restituisce un oggetto `WakeLockSentinel`, che permette di rilasciare manualmente il blocco oppure di reagire alla sua eventuale perdita tramite eventi dedicati. L'API prevede infatti che il sistema possa in alcuni casi revocare automaticamente il lock, ad esempio in seguito al blocco del dispositivo o al passaggio della scheda a uno stato inattivo, rendendo necessaria una gestione resiliente da parte dell'applicazione.
Dal punto di vista della standardizzazione, la Screen Wake Lock API è una specifica del W3C attualmente in stato di *Candidate Recommendation*, dunque in una fase avanzata del processo di definizione degli standard. La documentazione disponibile su MDN ne attesta la maturità: l'API è considerata sicura, richiede il contesto HTTPS per essere utilizzata e non è classificata come sperimentale. Il supporto nei principali browser è ampio e consolidato: risulta attiva sui browser basati su Chromium, su Firefox e su Safari, sia nella versione desktop che in quella per iOS. In tutti i casi in cui è essenziale che l'utente possa mantenere la visibilità costante del contenuto, come nel caso della navigazione assistita, della consultazione di istruzioni passo-passo, della scansione di codici o della supervisione di processi in tempo reale, la Screen Wake Lock API rappresenta uno strumento affidabile, maturo e pienamente integrabile nelle logiche di sviluppo web contemporanee [17].

2.6 Vantaggi e limiti di una PWA rispetto ad app native

Il principale punto di forza delle Progressive Web App (PWA) è la possibilità di **sviluppare un'unica app** che funzioni ovunque: su qualsiasi piattaforma o dispositivo. Questo garantisce sia un risparmio sia in termini di tempo che di risorse, poiché si tagliano i costi di sviluppo e la gestione del team diventa molto più semplice, siccome rimangono concentrati su un solo asset applicativo.
Come già detto, un ulteriore punto di forza è rappresentato dalla possibilità di funzionare anche in **modalità offline**. Una funzionalità particolarmente utile in contesti dove la connessione non è sempre garantita. Inoltre, quando vengono installate sulla schermata home del dispositivo, queste si mimetizzano perfettamente,

assumendo un aspetto simile ad un'app nativa.
Un esempio pratico di questo comportamento è rappresentato dall'applicazione **Trivago**. Nelle Figura 2.2, notiamo come, una volta installata sulla home (seconda immagine), la Web App perda la cornice del browser (la barra URL), acquisendo l'aspetto full-screen dell'app nativa (terza immagine), confermando la quasi indistinguibilità a livello di user interface, grazie all'adattamento progressivo al sistema. [2].

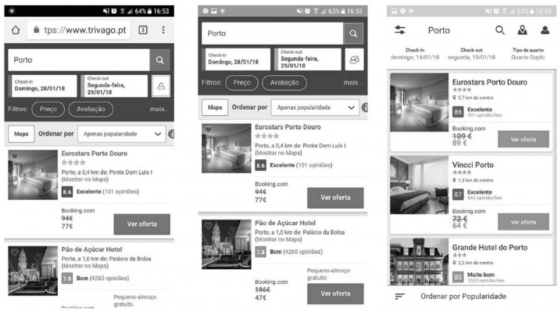

Figura 2.2: PWA vs App nativa (ottenuta da screenshot su Android)

Tuttavia, le PWA sono generalmente **meno veloci** delle app native, soprattutto in termini di user experience. Uno studio di W. Jobe ha analizzato la sostituibilità delle web application rispetto alle app native, sviluppando due prototipi (uno per il monitoraggio delle gare e uno per la prenotazione). Dai sei mesi di test è emerso che le performance della web app di monitoraggio erano compromesse dalla scarsa qualità del segnale GPS, mentre l'app di prenotazione ha registrato risultati comparabili a quelli della versione nativa, conseguendo pienamente gli obiettivi prefissati. [2].
Le PWA si stanno rivelando sempre più adatte anche in ambito sanitario. In uno studio recente, è stata sviluppata una Progressive Web App per l'autodiagnosi dei pazienti, e i risultati hanno evidenziato un'ottima soddisfazione da parte degli utenti, grazie alla velocità e all'organizzazione dell'applicazione. Questo conferma come le PWA possano rappresentare una soluzione valida e sempre più affidabile in diversi settori.[2].
Per l'utente finale, un beneficio aggiuntivo è la garanzia di un accesso costante alla release più aggiornata dell'applicazione, senza doverla scaricare o aggiornare manualmente tramite un app store. Per l'azienda, invece, sviluppare un unico prodotto velocizza l'introduzione di nuove tecnologie, semplifica il debugghing e, in generale, permette di rilasciare gli aggiornamenti in molto meno tempo rispetto alle app native separate.[2].
Le app native, al contrario, offrono prestazioni migliori, hanno accesso completo a tutte le funzionalità del dispositivo e sono spesso più sicure e stabili, permettendo

poi di creare un'esperienza su misura per quel sistema operativo. Tuttavia, questa metodologia, comporta un aumento dei costi, non solo a causa dello sviluppo multi-piattaforma, ma anche per la gestione di team e codebase paralleli. Ogni funzionalità richiede un'implementazione separata, con l'incognita di differenze di comportamento tra le piattaforme. Ciò induce spesso al rilascio sequenziale per minimizzare i rischi post-lancio, rallentando l'intero ciclo di vita del prodotto e perdendo opportunità di business.
Per un'azienda con budget limitato o per progetti orientati alla validazione di un'idea, le PWA rappresentano una soluzione ideale, capace di raggiungere il più vasto pubblico possibile con costi contenuti e ottimi risultati in termini di funzionalità e accessibilità.
Anche se le PWA hanno numerosi lati positivi, la loro diffusione è frenata da ostacoli tecnici, commerciali e di percezione pubblica. L'elemento più critico è la diversa **compatibilità tra browser**. Browser come Chrome offrono un supporto robusto alle funzioni PWA, ma altri (come Safari) mantengono un supporto parziale, specialmente per elementi chiave come le le **notifiche push** e la **sincronizzazione in background**. Queste discrepanze rendono l'esperienza utente incostante e rendono rischioso sviluppare app web avanzate per tutti.
Un altro limite è l'**accesso ridotto all'hardware del dispositivo**. A differenza delle app native, le PWA funzionano 'dentro' al browser e quindi non possono ancora accedere completamente ad alcune funzioni del sistema, come il Bluetooth, l'NFC o altre interfacce hardware avanzate (soprattutto su alcuni sistemi operativi). Questa limitazione restringe la possibilità di creare app complesse che hanno bisogno di un dialogo profondo con il dispositivo.
Dal punto di vista del pubblico, le PWA soffrono di una forte carenza di consapevolezza: la maggioranza degli utenti non è a conoscenza della loro esistenza o del loro valore. Questo è un fattore che alimenta la sfiducia, specialmente in merito alla **sicurezza**, dato che la mancata distribuzione tramite App Store ufficiali le rende meno affidabili. Infine, la mancanza di un luogo centralizzato di promozione ostacola la loro visibilità, rendendo la loro scoperta e la crescita molto più difficili per chi non ha già un nome affermato e una base utenti già consolidata.
Un altro punto dolente è la **monetizzazione**. A differenza delle app native, che hanno strumenti integrati per pagamenti e abbonamenti in-app, le PWA si trovano con soluzioni ancora poco sviluppate e spesso non standard. Non essendoci un'infrastruttura di pagamento unificata, per gli sviluppatori diventa un lavoro più complesso garantire acquisti sicuri. Questo è un fattore che può frenare la crescita e la sostenibilità finanziaria di molte applicazioni web.
Infine, vi sono considerazioni legate alla **sicurezza informatica**. Sebbene le PWA debbano essere obbligatoriamente servite tramite protocollo HTTPS, rimangono esposte a tutte le vulnerabilità tipiche delle applicazioni web, tra cui attacchi

XSS, CSRF e altre forme di compromissione lato client. La natura aperta dell'ambiente web impone dunque un livello particolarmente elevato di attenzione nella progettazione delle misure di protezione.

Capitolo 3

Autenticazione moderna: Standard e Tecnologie

3.1 Problemi della classica autenticazione con password

Nel contesto cybersecurity, l'autenticazione è critica per la tutela delle identità digitali e degli asset informativi, perchè permette di difendere chi siamo online e proteggere i nostri dati. Ma con la crescita dei servizi digitali, gli attacchi ai meccanismi di accesso aumentano a dismisura, rivelando un'allarmante necessità di strategie più resilienti. Per anni, le password sono state la nostra unica difesa. Oggi, però, queste sono troppo deboli e vengono aggirate facilmente da hacker esperti e tecniche di manipolazione sociale.

Sebbene l'autenticazione basata sul fattore **knowledge** (password) sia teoricamente robusta, in quanto il segreto dovrebbe risiedere unicamente nell'utente, la sua implementazione pratica è compromessa dalla negligenza umana. Infatti, Le persone scelgono password facili da ricordare e spesso le usano su tanti siti diversi, senza mai cambiarle. Questo le espone a rischi come il **phishing** o il furto di dati. Anche usare i password manager o inventare password complicate non risolve del tutto il problema; semplicemente, rende la vita più difficile all'utente. Questa inefficienza ha una rilevanza economica significativa: le stime indicano che il costo per ripristinare una credenziale dimenticata si aggira intorno ai 70 dollari, una cifra che si moltiplica in un flusso annuale da oltre un milione di dollari per le grandi imprese [18].

La piattaforma specializzata **Auth0** evidenzia che, nonostante la loro funzione originaria, le password sono diventate uno dei principali punti deboli dell'autenticazione. Le vulnerabilità tipiche includono vettori di ingegneria sociale, credential

harvesting tramite phishing, la possibilità di riproduzione remota (replay) del token di credenziali e l'inadeguatezza della gestione crittografica server-side. Anche l'integrazione di fattori biometrici presenta una problematica di irreversibilità del data leak. Non a caso, Forbes le ha etichettate come "l'elemento più vulnerabile", il punto dove gli aggressori sferrano i loro attacchi più frequenti come il credential stuffing o l'account takeover (ATO) [19] [20] [21].

Alla luce di queste criticità, l'industria ha sollevato lo sguardo verso orizzonti più sicuri: sistemi che intrecciano più prove d'identità o che, audacemente, cancellano il concetto stesso di segreto testuale. L'**Autenticazione Multifattoriale (MFA)**, in particolare, combina elementi relativi a knowledge, possession ed inherence, massimizzando l'accuratezza nell'identificazione dell'utente. Ciononostante, l'implementazione di tali framework introduce nuove sfide operative, tra cui problemi di usabilità, vincoli sull'infrastruttura hardware di accesso e una gestione meticolosa della privacy e della conformità normativa.

3.1.1 Phishing

Il phishing rappresenta una delle minacce più pervasive ed efficaci contro i meccanismi basati su password. L'attacco sfrutta la componente umana, sfruttando strategie di ingegneria sociale mirate a manipolare l'utente, inducendolo all'immissione volontaria delle proprie credenziali su interfacce che mimano con alta fedeltà quelle originali. È proprio questa fedeltà di clonazione delle interfacce utente autentiche che rende i phishing site difficilmente distinguibili, vanificando spesso anche gli sforzi degli utenti più avanzati. Il risultato è il furto di password e l'accesso non autorizzato agli account, un problema reso ancora più grave dal fatto che molti usano password banali o le stesse per tutti i servizi.

I report di sicurezza attestano che più del 70% delle operazioni di phishing è esplicitamente finalizzato all'acquisizione di coppie username/password, evidenziando la loro criticità strategica per gli attori malevoli. La situazione è peggiorata dalle nostre abitudini: circa il 30% delle persone usa password deboli (facili da indovinare), e il 58% le ripete su più siti. Questa tendenza al riutilizzo amplifica il vettore di rischio, convertendo la violazione di un'unica piattaforma in una potenziale compromissione multi-account. [22] [23].

Per contrastare il problema, una delle tattiche più efficaci è usare delle credenziali non-persistenti, come le One-Time Passwords (OTP). Queste chiavi effimere, generate algoritmicamente per sessione e trasmesse su canali esterni, riducono drasticamente l'esposizione al credential stuffing. Un altro aiuto fondamentale arriva dai gestori di password integrati. Questi strumenti verificano automaticamente l'URL prima di inserire le credenziali, bloccando l'attacco phishing e incoraggiando l'utente a creare password uniche e complesse per ogni servizio. [24].

Un ulteriore livello di protezione è fornito dall'applicazione di tecniche crittografiche come *hashing* e *salting* per la conservazione delle password lato server. L'hash consente di memorizzare una rappresentazione non reversibile della password, mentre il sale, ovvero un valore casuale unito alla password prima del processo di hashing, impedisce l'utilizzo efficace di dizionari precomputati e attacchi a forza bruta. Tuttavia, sebbene queste tecniche proteggano la persistenza dei dati, esse non affrontano il problema alla radice, ossia l'acquisizione fraudolenta della password direttamente dall'utente tramite phishing.

3.1.2 Credential Stuffing e Account Takeover

Il furto di credenziali (Credential Stuffing) e l'appropriazione di un account (Account Takeover) rappresentano due minacce fortemente collegate. Entrambe sfruttano le password deboli e l'abitudine degli utenti a usare la stessa password su tutti i siti. Sebbene distinti operativamente, i due attacchi si manifestano tipicamente in sequenza logica: il primo, che sfrutta liste di credenziali precedentemente compromesse, costituisce la condizione necessaria per la successiva compromissione dell'account, culminando nella realizzazione dell'ATO.
Il Credential Stuffing è un metodo d'attacco automatizzato che utilizza liste di nomi utente e password rubate (da un data breach) per provare ad accedere ad altri siti. Questo approccio permette agli attacker di ottenere accesso non autorizzato a endpoints multipli senza necessità di conoscere la vittima, ma semplicemente "riciclare" le informazioni sottratte. l processo è interamente mediato da agenti bot che possono fare milioni di tentativi in poco tempo, a volte riuscendo anche a superare i controlli di sicurezza come i CAPTCHA.
L'ATO rappresenta il culmine della violazione, in cui l'attacker stabilisce la persistenza e il controllo operativo sull'account utente. Questo tipo di attacco, non può avvenire solo tramite Credential Stuffing, ma anche tramite phishing, virus (malware) che rubano i dati, o con attacchi meno sofisticati (brute force).
Le conseguenze di questi attacchi sono molto serie, sia per le singole persone che per le grandi aziende. Un esempio notevole è il caso di Canva, dove i dati di circa 139 milioni di utenti sono stati rubati. I problemi che ne derivano sono concreti: perdite finanziarie (furti di denaro), danni all'immagine pubblica e un grave calo della fiducia dei consumatori. Tale fragilità diventa catastrofica nei luoghi dove il denaro e i segreti sono custoditi, come banche, assicurazioni, e-commerce, rendendo la sicurezza dell'identità una questione di sopravvivenza [25].
La mitigazione di questi attacchi richiede un approccio su più livelli che metta insieme la responsabilità dell'utente e la tecnologia avanzata. Tali soluzioni includono l'implementazione dell'Autenticazione Multifattoriale (MFA), l'analisi comportamentale (Behavioral Analytics) per notare accessi anomali e sistemi di monitoraggio istantaneo che identificano e fermano gli attacchi automatici di credential

stuffing. La realtà ha smentito l'efficacia della singola password; siamo costretti a un passaggio epocale verso architetture che siano più intelligenti e indistruttibili.

3.2 Autenticazione passwordless

Verificare l'identità di un utente senza usare le password è il principio base dell'autenticazione passwordless. Le metodologie prevalenti includono la conferma del fattore di possesso (attraverso la validazione di un device registrato o di un secondary account) e l'utilizzo di fattori biometrici. Questi ultimi possono essere tratti fisiologici (come l'impronta) o comportamentali (come il timbro della voce o la firma digitale).
L'autenticazione senza password è un grande vantaggio per le aziende: riduce i costi e i rischi di sicurezza. Eliminare completamente le password significa anche eliminare la possibilità di violazioni legate al loro furto. Questo approccio neutralizza l'efficacia del Credential Stuffing, rendendola di fatto non attuabile. Senza password, si impedisce ai criminali informatici di utilizzare credenziali rubate per accedere agli account di sistema.
L'utilizzo di metodi di autenticazione contemporanei come dispositivi conformi allo standard FIDO per l'autenticazione senza password riduce la sicurezza delle organizzazioni anche agli attacchi di phishing. In media, un individuo ha 100 password da ricordare e impiega 12,6 minuti per reimpostarle ogni settimana. Come risultato, le organizzazioni devono spendere più soldi per reimpostare le password e dedicare più tempo al servizio clienti. Poiché gli utenti potranno accedere senza password, l'autenticazione senza password può contribuire a ridurre o eliminare questi costi. In questo modo, anche la gestione e l'archiviazione dei database di password viene eliminata. Poiché quasi la metà degli oltre 1.700 professionisti IT intervistati ha dichiarato di non essere riuscita a completare una transazione personale a causa di una password dimenticata, l'eliminazione delle password potrebbe aumentare le vendite di alcune aziende, secondo una ricerca condotta dal Ponemon Institute e da Yubico [20].

3.2.1 Tipi di autenticazione senza password

Il processo di autenticazione è basato sull'uso dei fattori di autenticazione, ovvero prove che dimostrano l'identità di un utente. Questi vengono divisi in tre categorie: Conoscenza (qualcosa che l'unente conosce, come le password), Possesso (qualcosa che l'utente possiede, come un dispositivo o un email) e Essenza (ciò che l'utente è, come un tratto biometrico). I vecchi sistemi usano il primo tipo, cioè la password. I nuovi sistemi senza password si concentrano invece sul provare di avere un oggetto o di essere una persona specifica (ad esempio con l'impronta digitale), rendendo l'accesso molto più difficile da falsificare.

Queste sono alcune delle tecniche più comuni utilizzate per verificare i fattori di possesso e eredità:
Biometria: caratteristiche fisiche uniche per verificare l'identità senza bisogno di password. Poiché ogni persona è unica, l'autenticazione usa le differenze dei tratti fisici per confermare l'identità. L'efficacia di tecniche come il riconoscimento facciale è quasi assoluta: le statistiche ci dicono che la probabilità che due volti siano identici è inferiore a una su un trilione.
Link magici: rappresentano una modalità di autenticazione passwordless basata sul fattore di possesso. L'utente avvia il processo con l'input dell'indirizzo email nel campo di login. Viene poi generato e inviato un URL temporaneo che, una volta cliccato, completa l'autenticazione. Questa procedura viene ripetuta ad ogni nuovo accesso, garantendo che solo chi possiede quella casella di posta possa entrare.
Password o codici utilizzabili una volta: funzionano in modo simile ai link magici, ma la verifica avviene diversamente: l'utente riceve un codice numerico temporaneo sul cellulare via SMS o tramite e-mail. Invece di cliccare, deve digitare questo codice per accedere. Questo processo viene ripetuto per ogni nuovo accesso.
Notifiche in tempo reale: Gli utenti possono accedere a un'app di autenticazione dedicata, come le app SPID, tramite una notifica push sui loro dispositivi mobili e accedere all'app per verificare la loro identità.

3.3 Autenticazione Biometrica

L'autenticazione biometrica è un sistema di sicurezza innovativo che usa le diverse caratteristiche fisiologiche e comportamentali per garantire e validare l'accesso. Sfrutta dati come impronte, tratti del viso, voce o la scansione dell'iride. Tali identificatori sono intrinsecamente legati al soggetto, rendendo la loro duplicazione estremamente difficile. I login biometrici sono fluidi e veloci, offrendo un grado di sicurezza che dissolve i timori legati alle password violabili e ai token smarriti. Nonostante i suoi vantaggi, la sicurezza biometrica presenta alcuni problemi, quali la privacy dei dati, le preoccupazioni etiche e la possibilità di spoofing, che richiedono algoritmi robusti e attenta considerazione. [26]

3.3.1 Tipi di autenticazione biometrica

Le caratteristiche usate per l'autenticazione biometrica sono divise in due categorie: fisiologiche (come le impronte o l'iride) e comportamentali (come la voce o il modo di camminare).
Tra tutte, **l'impronta digitale** è una delle più vecchie e resta la più usata, specialmente sugli smartphone per l'accesso rapido e nei sistemi di controllo. Utilizzando metodi di scansione ottica, capacitiva e ultrasuoni, identifica le persone analizzando i disegni e le creste distintive presenti sulle dita. I suoi vantaggi e utilizzi includono

una elevata precisione e sensori compatti. Tuttavia, presenta potenziali vulnerabilità ai tagli e usura delle dita, nonché la possibilità di spoofing con impronte false.
Un'ulteriore tecnica è il **riconoscimento dell'iride**. Gli intricati disegni dell'anello colorato che circonda la pupilla possono essere riconosciuti utilizzando imaging a infrarossi per la cattura. Rinomato per la sua precisione e resistenza ai cambiamenti delle condizioni esterne, viene utilizzato nei sistemi di sicurezza alle frontiere per identificare i viaggiatori, ma può anche essere incorporato negli smartphone o nei visori AR per consentire un accesso sicuro. Tuttavia, l'elevato costo implementativo e i vincoli di usabilità legati alla vicinanza al sensore limitano la sua diffusione su larga scala.
Il **riconoscimento vocale** è un'opzione di autenticazione usata spesso nei call center per verificare i clienti e nei dispositivi domestici intelligenti per i comandi. Ciò viene raggiunto attraverso l'analisi tempro-spettrale delle caratteristiche vocali, come intonazioni, tono e ritmo. È senza dubbio vantaggioso perché è senza contatto e facile da integrare con i dispositivi mobili. Tuttavia, la sua sicurezza è compromessa da rumori di fondo e dalla capacità di qualcuno di simulare la voce di un altro.
L'uso del **riconoscimento facciale** è particolarmente noto negli ultimi anni, principalmente grazie alla sua integrazione nei telefoni cellulari come metodo di accesso rapido. Questa tecnica usa sensori (a infrarossi o telecamere) per una mappatura geometrica del volto (2D/3D), analizzando dettagli come la distanza tra gli occhi. Vantaggioso per la sua non-invasività, come nell'uso della videosorveglianza, incontra limiti operativi legati a condizioni luminose non ottimali e alterazioni morfologiche (invecchiamento). Inoltre, solleva questioni di privacy e rischi di bias algoritmico, che richiedono particolare attenzione in fase di implementazione.
Oltre ai metodi più conosciuti, stanno emergendo tecniche meno comuni con usi particolari. Il **riconoscimento dell'impronta palmare** sfrutta i disegni complessi delle vene all'interno della mano. Poiché queste vene sono difficili da replicare e non sono esposte, questa tecnica offre una sicurezza molto alta contro i tentativi di falsificazione.
Un'altra opzione è l'**analisi dell'andatura**, che classifica le persone dal loro modo di camminare. Anche non essendo una tecnica molto precisa, è utile in contesti come la videosorveglianza dove l'identificazione deve avvenire da lontano e con pochi dati disponibili.
Infine, un'altra tecnica è il **riconoscimento della forma dell'orecchio**, che basa sull'analisi della geometria unica del padiglione auricolare. Questa caratteristica è stabile nel tempo e può essere usata come prova aggiuntiva di autenticazione in situazioni specifiche.[26]

3.3.2 Processo di autenticazione biometrica

La verifica biometrica si svolge in quattro passaggi chiave. Inizialmente, i sensori (microfoni, telecamere) acquisiscono i dati. La precisione del riconoscimento è direttamente influenzata dalla fedeltà del campione di dati acquisito: un'acquisizione accurata riduce i Falsi Positivi e Rejection Rates. Poi, il sistema analizza e isola le caratteristiche (per un'impronta, i punti dove le linee si dividono) per trasformarle in una formula matematica unica. Infine, algoritmi avanzati confrontano questo modello con quelli già salvati, stabilendo la corrispondenza tramite la valutazione di metriche di somiglianza. Quindi, per una fase di verifica, è necessario avere un modello di riferimento in memoria, che deve essere stato acquisito e memorizzato nella fase di registrazione. Per garantire l'identità durante l'uso successivo, viene quindi condotto un confronto uno-a-uno.[26]

3.3.3 Problemi, implicazioni etiche e metriche per i sistemi biometrici

Adottare sistemi biometrici è fondamentale per la sicurezza, ma crea anche importanti sfide tecniche ed etiche.
La minaccia principale sono gli **attacchi di spoofing**: i criminali usano repliche ingannevoli (come maschere o registrazioni) per riuscire a entrare nel sistema. La mitigazione si attua tramite l'uso di tecniche anti-falsificazione: l'implementazione del liveness detection (rilevamento della vivacità), che assicura la prova di presenza del soggetto vivo, o l'adozione di sistemi multimodali che aumentano il fattore di sicurezza attraverso l'aggregazione di più modalità biometriche.
La **violazioni dei dati biometrici** è un altro grave problema di vulnerabilità. La netta distinzione dalle password risiede nella natura permanente e insostituibile delle caratteristiche umane. Una loro compromissione genera un impatto disastroso e senza possibilità di recupero per la sicurezza dell'utente. Per controllare questa minaccia, è cruciale che le organizzazioni utilizzino meccanismi di crittografia rigorosi per blindare i dati sia quando sono conservati nei database, sia nel momento della loro trasmissione tra i sistemi.
L'efficacia della biometria è spesso limitata anche da fattori esterni, come **l'ambiente**. Il riconoscimento del volto è un esempio di come la scarsa illuminazione possa degradare le prestazioni. Per affrontare queste limitazioni, si usano soluzioni basate sul machine learning, consentendo ai modelli di adattarsi e funzionare bene anche in condizioni che cambiano.
Però, l'uso della biometrica solleva anche **questioni etiche** importanti. La riservatezza dei dati rappresenta il nodo focale: data la natura delle informazioni personali raccolte, il rischio di uso improprio o di violazioni non autorizzate è elevato. È necessario stabilire normative rigorose sulla gestione dei dati, affiancate

da meccanismi di consenso informato e tecniche di anonimizzazione per bilanciare sicurezza e tutela dei diritti individuali.
La presenza di **disparità e pregiudizi** nei sistemi biometrici è un secondo problema, che potrebbe derivare da set di addestramento inadeguati. In questi casi, il sistema funziona in modo incoerente e penalizza alcuni gruppi di persone in base a fattori come etnia, genere o età. Per evitare queste distorsioni, è necessario che i dataset siano ampi, diversificati e realmente rappresentino le varie popolazioni. Inoltre, è necessario che i sistemi vengano sottoposti a verifiche periodiche allo scopo di identificare e correggere qualsiasi bias.
L'aspetto più attuale è l'uso etico della biometria, in particolare per limitare la sorveglianza e il monitoraggio senza consenso. Per gestire questo aspetto delicato, è essenziale stabilire regole di condotta etica vincolanti per l'utilizzo dei sistemi, affiancate da una trasparenza completa su come operano e sulle motivazioni precise che ne guidano l'installazione. [26]

3.4 Web Authentication API

Web Authentication API (WebAuthn) è un'estensione della Credential Management API [1] e si fonda sull'uso della crittografia a chiave pubblica, offrendo così un'autenticazione forte, senza password e resistente agli attacchi più comuni. [27]
Questa tecnologia si fonda sulla creazione di due chiavi complementari. La chiave privata è custodita in modo sicuro e non lascia mai l'autenticatore (authenticator), che è il dispositivo hardware o software dell'utente. In netto contrasto, la chiave pubblica è l'informazione necessaria per la verifica e viene depositata in modo sicuro sul server di destinazione. Per l'autenticazione, invece di inviare una password o un codice, l'utente dimostra di possedere la chiave privata firmando digitalmente una "sfida" inviata dal sito. Qualora l'autenticatore integri un sistema di verifica locale, come la biometria, la firma è condizionata al superamento del controllo di identità. Questo processo aggiunge un importante strato di protezione, in quanto i dati biometrici cruciali non vengono mai trasmessi, ma servono solo a sbloccare l'autorizzazione all'uso della chiave sul dispositivo. [27] [28]
I dati firmati provengono da due insiemi principali: i client data e gli authenticator data.I dati firmati provengono da due insiemi principali: i client data e gli authenticator data. I primi includono, tra le altre informazioni, il dominio (origine) del sito a cui si sta accedendo e la challenge generata dal server, garantendo che l'autenticazione sia collegata al contesto corretto e riducendo significativamente i rischi

[1] Le Credential Management API sono strumenti che permettono ad un sito web di creare, archiviare e recuperare le credenziali degli utenti, come password, token di sicurezza e chiavi SSH.

di phishing. Gli authenticator data, invece, contengono informazioni aggiuntive generate dal dispositivo, che rafforzano la validità del processo. Il server non si limita a verificare la firma, ma conferma anche la correttezza della challenge, impedendo così possibili attacchi di replay. I benefici di questa soluzione sono numerosi. Oltre a garantire la difesa contro il phishing, riduce il danno causato dalle violazioni di dati: se rubano la chiave pubblica, la sicurezza non viene compromessa, poiché è indispensabile la chiave privata per autenticarsi. Di conseguenza, il sistema assicura un livello di sicurezza notevolmente superiore unito a una maggiore semplicità d'uso per l'utente, risolvendo i problemi cronici delle password, come la tendenza al riutilizzo e la vulnerabilità agli attacchi mirati [27] [28].

3.4.1 Processo di registrazione

Il processo di registrazione viene fatto una sola volta per collegare un autenticatore al proprio account. In questa fase, il dispositivo crea la coppia di chiavi crittografiche: la chiave pubblica viene inoltrata al server, che la registra come elemento di accesso, mentre la chiave privata è custodita in modo inalterabile all'interno del dispositivo e non può essere estratta. I dati inviati, inclusi la challenge e gli authenticator data, sono firmati crittograficamente dall'autenticatore. Solo dopo la validazione positiva della firma, la chiave pubblica e i metadata sono accettati dal server come nuove credenziali persistenti.
L'operazione di registrazione si distingue perché include l'invio della nuova chiave pubblica, che servirà da riferimento per ogni autenticazione futura. Il sistema si basa sul protocollo challenge-response (sfida-risposta), che garantisce che il server non solo verifichi la firma, ma anche che essa sia stata prodotta in risposta a una sfida unica e non falsificabile. In tal modo, l'utente può provare il possesso della chiave privata senza mai renderla visibile, proteggendosi attivamente dagli attacchi di replay.[27] [28]

Figura 3.1: Attori principali nel flusso di registrazione/autenticazione

Il processo di registrazione si svolge grazie all'azione coordinata di tre elementi fondamentali. L'autenticatore è il dispositivo che genera le chiavi segrete e appone le firme; il browser funge da intermediario che orchestra lo scambio di informazioni tra il dispositivo e il server tramite WebAuthn; e il server è l'organizzatore che prepara la sfida, riceve la convalida firmata e memorizza le credenziali. Il browser ha un ruolo chiave: garantisce che i dati inviati al dispositivo siano collegati al sito

corretto (il dominio), proteggendo la privacy dell'utente e impedendo frodi da siti malintenzionati. [28]
Una volta completata questa fase di registrazione, l'utente può accedere nuovamente utilizzando l'autenticatore che ha configurato in precedenza senza utilizzare la password, eliminado i pericoli associati alle password tradizionali.

3.4.2 Esempio pratico

Per comprendere meglio il funzionamento della registrazione in WebAuthn, si potrebbe pensare a un caso in cui un utente, che chiameremo John, desidera configurare il lettore di impronte digitali del proprio smartphone come autenticatore.
L'avvio del processo è gestito dal server, che genera una sfida crittografica, un numero casuale e irripetibile (nonce) che vale unicamente per l'autenticazione in corso e viene subito scartato. La sfida è associata all'identificativo dell'utente e inviata al client (browser) insieme a metadati critici, quali l'User ID (identificatore utente), per stabilire il contesto di autenticazione.
A questo punto, l'applicazione web usa l'API di autenticazione WebAuthn, in particolare il comando `navigator.credentials.create( publicKey: ... )`, per richiedere la creazione di una nuova credenziale. La richiesta veicola parametri come il nonce della challenge, l'identità dell'utente e gli algoritmi crittografici specificati. A ciò si aggiunge l'intervento del browser, che integra autonomamente il nome di dominio del sito; questo dettaglio tecnico assicura che l'intera operazione sia vincolata al sito legittimo e che sia impossibile sfruttare la procedura da parte di domini fraudolenti.
Una volta che il payload è pronto, viene inviato all'autenticatore, che chiede esplicitamente l'approvazione dell'utente (ad esempio, tramite l'impronta digitale). Solo dopo l'interazione positiva, l'autenticatore procede alla generazione crittografica della coppia di chiavi. La chiave pubblica è l'output che ritorna al client per la registrazione, mentre la chiave privata, associata al credential ID e ai metadati, è conservata in modalità non estraibile sul dispositivo.
Oltre alla chiave pubblica, questa credenziale contiene una firma crittografica, il dominio, il credential ID e altri parametri tecnici. Pertanto, l'applicazione invia questi dati al server, che verifica la validità della firma per garantire che la risposta sia vera e propria. Una volta completata la verifica, il server memorizza la chiave pubblica e il credential ID per collegarli in modo permanente all'account dell'utente. Inoltre, il server invalida la sfida crittografica, rendendola impossibile da riutilizzare in un attacco di tipo replay.
Al termine della fase di registrazione iniziale, il flusso fornisce all'utente una credenziale sicura che è collegata alla sua identità e controllata dal lettore biometrico del dispositivo [28].

3.4.3 Verifica e Attestazione

Appena l'applicazione web ha avviato il processo, l'autenticatore confronta scrupolosamente l'indirizzo del sito che sta richiedendo l'accesso con quello che ha registrato in memoria. Questo meccanismo di confronto del dominio protegge il sistema dai tentativi di phishing, poiché un sito malevolo non può superare il cross-origin check (controllo tra origini). Se i domini sono gli stessi, il dispositivo verifica l'identità dell'utente in locale (ad esempio, tramite sensore biometrico). Ottenuto l'esito positivo, la sfida e il dominio vengono firmati con la chiave privata, autorizzando l'accesso.
A questo meccanismo di autenticazione si affianca una questione ulteriore, ossia quella relativa al processo di registrazione di una nuova credenziale. Al momento della creazione della chiave pubblica in questo caso, è possibile chiedere quale chiave privata venga utilizzata per firmare i dati dell'autenticatore e del cliente. La risposta risiede nell'uso di una chiave privata unica e specifica per ciascun modello di autenticatore, integrata nel dispositivo stesso in fase di produzione. Per verificare tale firma, il server deve avere a disposizione la chiave pubblica corrispondente. Questa può essere preconfigurata sul server per dispositivi già noti e considerati affidabili, oppure ottenuta da registri pubblici come il FIDO Metadata Service, che mette a disposizione le chiavi pubbliche relative ai diversi modelli di autenticatore. Questa procedura è definita Attestazione e costituisce una garanzia supplementare, consentendo di accertare che l'autenticatore che si sta registrando sia effettivamente un modello certificato e legittimo. L'Attestazione si formalizza in una struttura dati nota come Attestation Statement, che contiene informazioni di identificazione del modello ed è firmata con la chiave privata del dispositivo. Nonostante i vantaggi in termini di sicurezza, questo processo solleva preoccupazioni significative per la privacy: la divulgazione del modello dell'autenticatore rende l'utente identificabile e crea il potenziale per un tracciamento non etico.
Per questo motivo, nella configurazione predefinita, il client scarta un'attestazione senza inviarla al server, anche se l'autenticatore la restituisce. Infatti, per bilanciare le esigenze di sicurezza con quelle di protezione della privacy, l'attestazione viene condivisa solo se esplicitamente richiesta come opzione e con il consenso dell'utente finale [27] [28].

Capitolo 4

OAuth2.0 e Architetture di autenticazione

4.1 OAuth 2.0

OAuth 2.0 è un framework di autorizzazione popolare che consente a terze parti di accedere a servizi HTTP limitati senza che gli utenti debbano condividere le proprie credenziali. Adottato da grandi aziende come Google, Facebook e Microsoft, è diventato lo standard per la sicurezza delle API. Il suo funzionamento è basato sui token di accesso, che consentono un equilibrio tra flessibilità e sicurezza nei processi di autenticazione concedendo autorizzazioni limitate a particolari risorse. Per avviare il processo, l'utente normalmente clicca su un pulsante o su un link all'interno di un'applicazione client. Questo lo porta al server di autorizzazione, dove viene inviata una richiesta che indica a quale risorsa l'applicazione desidera accedere. Se l'utente consente, il server verifica la sua identità e crea un token di accesso. Tale token viene successivamente inviato in modo sicuro all'applicazione client e utilizzato per interagire con le risorse protette ospitate sul server web.
OAuth 2.0 è più flessibile rispetto alla prima versione e non è un protocollo rigido. Invece, è un framework estremamente flessibile che supporta una varietà di tipi di grant e token. Questa funzione lo rende adatto a una vasta gamma di scenari applicativi, inclusi applicazioni web tradizionali, dispositivi mobili e IoT.
L'adozione di OAuth 2.0 porta con sé diversi vantaggi. Poiché gli utenti non sono più costretti a condividere le proprie credenziali con applicazioni di terze parti Allo stesso tempo, fornisce all'utente più influenza, consentendogli di scegliere in modo approfondito quali risorse autorizzare e per quanto tempo. Il suo carattere flessibile lo rende adattabile a una varietà di contesti, mentre il suo essere un framework diffuso e aperto promuove l'interoperabilità e la standardizzazione tra diverse piattaforme e servizi. [29]

4.1.1 Token in OAuth 2.0

I token sono una parte essenziale del protocollo OAuth 2.0 per gestire sicuramente le autorizzazioni. Un token è una stringa che un server di autorizzazione ha firmato per garantire che le informazioni che contiene siano sicure. Dettagli sull'utente come autorizzazioni, gruppi di appartenenza e indicatori temporali sono inclusi in queste informazioni. Finché il token è considerato valido, l'accesso dell'utente alle risorse protette rimane disponibile.
OAuth 2.0 si basa principalmente su due tipi di token: i token di accesso (Access token) e i token di aggiornamento (Refresh token). I primi consentono l'accesso alle risorse protette, ma lo fanno per un breve periodo per ridurre i rischi di una compromissione. I secondi, al contrario, consentono di ottenere nuovi token di accesso senza richiedere nuovamente le credenziali dell'utente, il che consente sessioni più lunghe e un'esperienza d'uso più fluida.
Un token di accesso è generalmente composto da tre componenti: header, payload e signature. Il tipo di token e l'algoritmo utilizzato per la firma sono indicati nell'header; Le informazioni principali relative all'utente, alle autorizzazioni concesse e alle scadenze sono contenute nel payload, noto anche come claims section; Infine, la componente di verifica, la signature, garantisce l'autenticità del token utilizzando algoritmi crittografici e funzioni di hash difficili da contraffare.
Il payload, che costituisce il cuore del token, contiene informazioni importanti come i dati dell'utente o del cliente, gli ambiti di autorizzazione (scope), la durata di validità e gli identificatori associati all'applicazione. È stato progettato per mantenere dimensioni ridotte. Il JSON Web Token (JWT), una codifica comune in questo contesto, consente di rappresentare queste informazioni in modo efficace e compatto. Questa struttura consente l'utilizzo di un unico token per accedere a più applicazioni o servizi all'interno dello stesso ecosistema, come nei servizi Google, dove gli utenti possono condividere token con Drive, Gmail e Calendar.
L'adozione dei token in OAuth 2.0 presenta vantaggi notevoli. È possibile specificare con precisione i permessi, la durata e il grado di autorizzazione di ciascun token in termini di flessibilità. Dal punto di vista della sicurezza, è meglio evitare di inviare frequentemente informazioni di sicurezza come username e password per ridurre la portata degli attacchi. Inoltre, ogni token può essere revocato o scaduto automaticamente, il che consente un controllo tempestivo e riduce i danni in caso di compromissione. Infine, la possibilità di stabilire autorizzazioni specifiche consente di concedere solo le autorizzazioni necessarie, aumentando la protezione. [29]

4.2 Flusso di Autorizzazione

Il flusso astratto di OAuth 2.0 è suddiviso in più fasi e spiega come funzionano i quattro ruoli principali: client, resource owner, authorization server e resource

server. Tutto inizia quando il client richiede l'autorizzazione al proprietario delle risorse. Questo può avvenire direttamente o, più spesso, attraverso l'intermediazione dell'authorization server. Se l'utente concede i permessi, il client riceve un riconoscimento di autorizzazione, che è una credenziale che indica l'autorizzazione del proprietario delle risorse. Questo riconoscimento può essere espresso in uno dei vari tipi di riconoscimento previsti dalla specifica o tramite un'estensione.
Successivamente, il client presenta l'Authorization Grant (Autorizzazione Concessa) all'Authorization Server, identificandosi in modo sicuro per richiedere in cambio il token di accesso. Il server verifica l'autenticità del client e la validità del Grant e, solo in caso di esito positivo, rilascia il token, che verrà poi utilizzato dal client nelle successive richieste al Resource Server come Bearer Token. Ora, il client può inviare richieste al server delle risorse, che controllerà la validità del token prima di concedere l'accesso ai dati protetti e soddisfare la richiesta. [29]

4.3 JWT (JSON Web Tokens)

I JSON Web Tokens (JWT) sono uno standard aperto ampiamente utilizzato per garantire la trasmissione sicura di informazioni tra due parti; particolari applicazioni si trovano nei contesti di autorizzazione e autenticazione delle applicazioni web. La loro diffusione è principalmente dovuta alla loro struttura compatta, che può essere serializzata facilmente in un formato sicuro per URL e alla facilità con cui i sistemi che li ricevono possono validarli.
Dal punto di vista tecnico, un JWT rappresenta un insieme di claims, cioè affermazioni o informazioni relative a un'entità, come l'identità di un utente o i suoi privilegi, codificate in un oggetto JSON. Questo oggetto, chiamato JWT Claims Set, viene incluso come payload in una struttura di tipo JSON Web Signature (JWS) o JSON Web Encryption (JWE). Nel primo caso, i reclami sono firmati digitalmente o protetti mediante un codice di autenticazione del messaggio (MAC). Nel secondo caso, i reclami sono cifrati per proteggerli. È anche possibile creare un JWT annidato, noto anche come Nested JWT, in cui un token è racchiuso in un altro JWS o JWE. Ciò consente di utilizzare entrambi i meccanismi di cifratura e firma. [30]

4.3.1 Struttura di un JWT

La struttura di un JWT è sempre composta da più parti separate da un punto (.), ciascuna delle quali indica un elemento codificato in base64url. Il token, nella sua forma più diffusa, cioè quella compatta basata su JWS, si divide in tre parti: l'header, che descrive gli algoritmi crittografici applicati; il payload, che contiene i claims effettivi; e la signature, che garantisce l'integrità e l'autenticità dei dati. Questa rappresentazione consente la trasmissione di un unico token in contesti web,

come le intestazioni HTTP, mantenendo sicurezza, portabilità e interoperabilità allo stesso tempo. [30]

4.3.2 L'Header

L'header, chiamato anche JOSE header, contiene le informazioni sul token e i parametri necessari per interpretarlo correttamente. In particolare, queste dichiarazioni delineano se il token è firmato o cifrato, gli algoritmi utilizzati per la crittografia e, più in generale, il modo in cui il destinatario deve processare il token. La presenza di alcuni campi nell'header potrebbe essere richiesta, a seconda della tipologia di JWT. Ad esempio, i JWT cifrati contengono informazioni sui algoritmi utilizzati per cifrare le chiavi e i contenuti, ma tali informazioni non sono necessarie nei JWT non cifrati.
L'unico claim richiesto nell'header è "alg", che indica l'algoritmo utilizzato per firmare o decifrare il token. Questo campo deve assumere il valore none nel caso in cui non sia prevista alcuna protezione. L'header può contenere parametri opzionali come typ e cty oltre a questo campo essenziale. Il primo, tip, indica il media type del token ed è utile quando i JWT possono essere confusi con altri oggetti che presentano un header JOSE. Tuttavia, è molto raro usarlo e viene solitamente impostato al valore JWT quando è presente. [31]

4.3.3 Il Payload

Il payload è la sezione del JWT in cui vengono inseriti i dati principali dell'utente o il motivo per cui viene utilizzato il token. Anch'esso è strutturato come un oggetto JSON ed è in grado di contenere sia informazioni personalizzate che dichiarazioni standard. Non ci sono campi richiesti, ma alcune dichiarazioni hanno un significato universalmente accettato e sono note come **registered claims**.
Tra i più rilevanti possiamo trovare iss (issuer), che identifica chi ha emesso il token, sub (subject), che identifica il soggetto a cui si riferisce il token, aud (audience), che specifica i destinatari previsti, ed exp, che stabilisce la data oltre il quale il token non è più valido. Altri claims, come nbf (not before), iat (issued at) e jti (JWT ID), garantiscono la validità temporale e l'unicità del token, contribuendo a prevenire l'uso non autorizzato.
Oltre ai registered claims, i JWT possono includere public claims, registrati presso l'IANA per evitare collisioni di nomi oppure private claims, definiti personalmente dai produttori e dai consumatori del token per soddisfare esigenze specifiche. La maggior parte dei JWT combina registered claims con private claims, adattandosi così al contesto applicativo di riferimento. [31]

4.3.4 La firma

Sebbene sia possibile generare un JWT valido semplicemente utilizzando solo header e payload, in questo caso si tratta di un JWT non sicuro perché manca l'elemento essenziale che garantisce l'integrità del contenuto: la firma. In realtà, la firma garantisce la veridicità del token garantendo che le informazioni interne non siano state modificate. Quindi non impedisce a terzi di leggere le informazioni contenute nel payload, ma solo di modificarle senza essere scoperti. La validazione del token è il processo di verifica della firma. Quando tutte le condizioni nel payload e nell'header sono soddisfatte, un JWT è valido. Questo punto è fondamentale perché la validità non dipende solo dalla correttezza della firma ma anche dal rispetto di eventuali vincoli temporali o di destinatario indicati nei reclami, come exp o aud. Di conseguenza, un token firmato può essere giudicato invalido se, nonostante la sua firma corretta, risulta scaduto o non destinato al destinatario.
Come accennato in precedenza, la specifica JWT consente la creazione di token privi di firma, noti come unsecured JWT, quando l'algoritmo indicato nell'header è impostato a none. Tuttavia, in questi casi, la validazione si limita al controllo dei claims nel payload senza alcuna garanzia di integrità. Questo costituisce anche un vettore di attacco ampiamente noto, che consiste nello stripping della firma: un attaccante può rimuovere la firma da un token firmato e alterare l'header per indicare l'uso di none, rendendo il token un JWT non sicuro. Pertanto, è responsabilità di coloro che utilizzano i JWT implementare correttamente i controlli di validazione e definire accuratamente i requisiti di sicurezza necessari per il proprio ambiente.
I Signed JWTs introducono un ulteriore elemento rispetto agli unsecured JWT, ovvero la firma. Nella forma compatta del token, essa compare dopo l'ultimo punto, rappresentando la terza sezione del JWT. Sebbene il suo compito sia quello di garantire l'integrità del contenuto e l'autenticità del mittente, il modo in cui la firma viene calcolata e verificata dipende dall'algoritmo utilizzato.

4.3.5 JWT firmati e generazione della firma

Un algoritmo crittografico viene applicato ai dati nell'header e nel payload di un token per creare la parte della firma di un JWT. Gli algoritmi di firma possono essere simmetrici o asimmetriche. Il payload e l'header vengono combinati con una chiave segreta condivisa e sottoposti a una funzione di hash utilizzando il meccanismo HMAC (HS256). Solo coloro che possiedono la stessa chiave segreta possono verificare la firma. Invece, gli algoritmi a chiave pubblica, come RS256 (RSA) o ES256 (ECDSA), generano la firma utilizzando la chiave privata, mentre chiunque possiede la chiave pubblica corrispondente può verificare. Quando il token deve essere validato da più soggetti, questo metodo è particolarmente utile.
È utile osservare l'implementazione pratica attraverso librerie consolidate per

comprendere come vengono utilizzati i JSON Web Token in un applicativo. jsonwebtoken è uno dei token JavaScript più utilizzati, che offre metodi per generare, firmare e verificare i token.

HS256: HMAC con SHA-256
Nell'algoritmo HS256, la firma del token si esegue usando una chiave segreta condivisa. Ciò implica la necessità di una conoscenza condivisa della chiave tra l'entità emittente e l'entità che esegue la validazione. Tale meccanismo è preferibile in configurazioni one-to-one o in ambienti digitali dove la chiave può essere custodita con una sicurezza impenetrabile, riducendo significativamente la probabilità di una compromissione o esposizione.
La creazione di un token firmato con HS256 è illustrata nel seguente tratto di codice. Definito il payload, che stabilisce il contenuto informativo del token insieme alla chiave segreta, il token viene poi generato e la sua validità temporale è impostata a cinque secondi mediante l'invocazione della sign function (funzione di firma).

```
1  import jwt from 'jsonwebtoken';
2
3  const payload = {
4    sub: "1234567890",
5    name: "John Doe",
6    admin: true
7  };
8
9  const secret = 'my-secret';
10
11 const signed = jwt.sign(payload, secret, {
12   algorithm: 'HS256',
13   expiresIn: '5s'
14 });
```

Listing 4.1: Esempio di firma con HS256

La funzione verify verifica la firma e la validità temporale del token. Per evitare attacchi di tipo signature stripping, in cui un avversario potrebbe cercare di forzare il sistema ad accettare firme non valide, è fondamentale specificare esplicitamente l'algoritmo accettato.

```
1  const decoded = jwt.verify(signed, secret, {
2    algorithms: ['HS256'],
3  });
```

Listing 4.2: Verifica di un token con specifica dell'algoritmo utilizzato

Se il token in questo caso venisse verificato dopo più di cinque secondi dalla sua generazione, risulterebbe automaticamente invalido e la libreria solleverebbe un'eccezione.

RS256: RSA con SHA-256

Un metodo diverso basato sulla crittografia asimmetrica viene utilizzato dall'algoritmo RS256. In questo caso, una chiave privata viene utilizzata per firmare il token, mentre la chiave pubblica viene utilizzata per verificarlo. In scenari one-to-many, questo meccanismo è molto utile perché diversi attori devono essere in grado di validare i token senza crearli, mantenendo così il principio di separazione dei ruoli.

Una coppia di chiavi RSA deve essere creata prima di poter firmare e verificare un token con RS256. La libreria OpenSSL è un metodo comunemente utilizzato per ottenere sia un file con chiave privata che uno con chiave pubblica:

```
// Generazione della chiave privata a 2048 bit
openssl genpkey -algorithm RSA -out private_key.pem -pkeyopt
    rsa_keygen_bits:2048

// Derivazione della chiave pubblica dalla chiave privata
openssl rsa -pubout -in private_key.pem -out public_key.pem
```

Listing 4.3: Creazione delle chiavi con OpenSSL

Una volta generate, le chiavi possono essere lette e utilizzate all'interno del codice. La chiave privata è impiegata per firmare il token:

```
import jwt from 'jsonwebtoken';

const privateRsaKey = `
-----BEGIN PRIVATE KEY-----
MIIBVgIBADANBgkq...
-----END PRIVATE KEY-----
`;

const signed = jwt.sign(payload, privateRsaKey, {
  algorithm: 'RS256',
  expiresIn: '5s'
});
```

Listing 4.4: Utilizzo della chiave privata per firmare il payload

Invece, la verifica richiede solo la chiave pubblica. In questo modo, chiunque possieda la chiave pubblica può verificare la validità del token, ma solo coloro che possiedono la chiave privata possono crearne di nuovi:

```
const publicRsaKey = `
-----BEGIN PUBLIC KEY-----
MFwwDQYJKoZIhv...
-----END PUBLIC KEY-----
`;

const decoded = jwt.verify(signed, publicRsaKey, {
```

```
8   algorithms: ['RS256'],
9 });
```

Listing 4.5: Verifica di un token con la chiave pubblica

L'uso di chiavi asimmetriche introduce un livello di sicurezza superiore rispetto ad HS256, riducendo il rischio che una chiave compromessa consenta sia la validazione sia la generazione di nuovi token. [31]

4.4 OAuth2-Proxy

L'OAuth2-Proxy è un elemento essenziale per consentire l'autenticazione sicura e la gestione degli accessi in ambienti cloud e applicazioni web, perché permette a utenti e servizi di identificarsi usando provider esterni (come Google o Azure), rendendo l'accesso e la sicurezza più semplici da gestire, senza richiedere agli sviluppatori avanzate competenze specifiche.
Nato come software open-source, OAuth2-Proxy è specificamente progettato per la semplificazione dell'integrazione dei security protocols OAuth2 e OpenID Connect. Il proxy può essere configurato secondo due schemi operativi fondamentali, mostrati nella Figura 4.1: la modalità auth middleware e la modalità reverse proxy. Ogni modalità definisce un metodo diverso per posizionare il proxy nella rete in relazione alle applicazioni da proteggere.
OAuth2-Proxy si interpone direttamente tra l'utente e l'applicazione, nella mo-

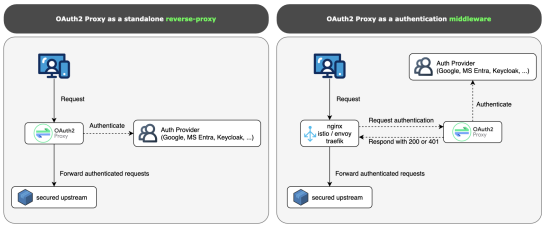

Figura 4.1: Reverse proxy e Auth middleware

dalità **Reverse Proxy**, fungendo da vero "gatekeeper" delle richieste in ingresso. Il proxy controlla prima il traffico diretto all'applicazione, e verifica lo stato di autenticazione dell'utente. In caso di mancata autenticazione, viene reindirizzato a un fornitore di identità come Google, per completare la procedura di login. Il proxy inoltra la richiesta iniziale all'applicazione backend, solo dopo aver ottenuto il token, e crea una sessione sicura, solitamente gestita tramite cookie criptati. L'applicazione, in questo modello, non gestisce direttamente la logica di autenticazione; invece, si basa sulle informazioni fornite dal proxy, attraverso header

HTTP specifici, contenenti i dati di identificazione dell'utente. Tale metodo è particolarmente vantaggioso in situazioni in cui si desidera aggiungere autenticazione a sistemi legacy, o a semplici web application che non supportano nativamente OAuth2 o OIDC, oppure in ambienti Kubernetes, dove il proxy può fungere da punto di controllo centralizzato del traffico.
Invece, la **modalità Middleware** di autenticazione consente un'integrazione meno invasiva. Il OAuth2-Proxy non riceve direttamente tutto il traffico, in questo caso. Invece, funge da servizio ausiliario e viene utilizzato spesso come sidecar, o come endpoint dedicato a cui il reverse proxy principale (come Nginx o Traefik) delega la verifica delle autenticazioni.
Il reverse proxy invia una sotto-richiesta a un endpoint specifico di OAuth2-Proxy, come /oauth2/auth, quando arriva una richiesta, per verificare se l'utente è già stato autenticato. In caso di approvazione, OAuth2-Proxy genera uno stato 200 e fornisce le informazioni necessarie; in caso contrario, restituisce un errore 401 o rimanda alla procedura di login. In questo caso, l'applicazione backend riceve solo richieste già verificate; tuttavia, il traffico deve passare attraverso l'OAuth2-Proxy.
Il confronto delle due modalità mostra le differenze importanti. Poiché è autosufficiente, e non richiede modifiche all'applicazione, il modello reverse proxy è più semplice da configurare. Tuttavia, in infrastrutture complesse dove l'intero traffico passa necessariamente dal proxy, è meno flessibile. Al contrario, la modalità Middleware richiede una configurazione e un coordinamento più approfonditi con altri elementi dell'architettura, ma garantisce una maggiore integrazione con sistemi più sofisticati e consente di mantenere un unico Reverse Proxy principale come punto di accesso.
Questo meccanismo chiarisce il ruolo dell'OAuth2-Proxy come gateway di autenticazione: il programma acquisisce e gestisce i token OAuth2 per conto dell'applicazione, avviando una sessione utente utilizzando cookie criptati. Poiché il Proxy riconosce il cookie e verifica la validità della sessione prima di consentire l'accesso e inviare la richiesta al Backend, l'utente non deve ripetere l'intero processo di login nelle richieste successive. Con questo metodo, è possibile integrare un sistema di autenticazione sicuro, senza dover modificare il codice dell'applicazione originale, richiedendo al proxy la gestione completa del flusso OAuth2.

4.4.1 Modalità di utilizzo e flusso operativo

L'installazione di OAuth2-Proxy può essere effettuata in vari modi, come un binario precompilato, un container Docker o un manifest Kubernetes. Prima di tutto, l'installazione e l'avvio del proxy vengono eseguiti, ad esempio scaricando il binario ufficiale o eseguendo l'immagine *quay.io/oauth2-proxy/oauth2-proxy* tramite Docker. Successivamente, per ottenere un Client Secret e un Client ID, è necessario creare e registrare un'applicazione OAuth presso l'Identity Provider preferito, come Google

o un altro IdP compatibile. In seguito, questi parametri, insieme alle informazioni sul provider e agli URL di reindirizzamento, devono essere configurati all'interno dell'OAuth2-Proxy utilizzando un file di configurazione, le opzioni a riga di comando o le variabili d'ambiente. Infine, il proxy viene solitamente posizionato dietro un endpoint HTTPS, come Nginx o un ingress controller Kubernetes, in modo che passi attraverso tutto il traffico dell'applicazione.
OAuth2-Proxy verifica innanzitutto se esiste già una sessione attiva mediante la presenza di un cookie quando un utente tenta di accedere all'applicazione protetta. In caso contrario, l'utente verrà reindirizzato alla pagina di login dell'Identity Provider. L'IdP reindirizza il flusso verso il percorso di callback del proxy (*/oauth2/callback*) dopo l'autenticazione e fornisce il codice di autorizzazione. A questo punto, OAuth2-Proxy termina il processo di assegnazione del codice di autorizzazione scambiando il codice per i token di accesso e creando un cookie di sessione sicuro. D'ora in poi, ogni richiesta che include questo tipo di cookie viene riconosciuta e inoltrata al backend senza che l'utente debba autenticarsi nuovamente. Dal punto di vista delle applicazioni sottostanti, il proxy funge da client OAuth2 e gestisce completamente il protocollo con l'Identity Provider. Inoltre, può migliorare le richieste verso il backend inserendo intestazioni HTTP aggiuntive come *X-Auth-Request-Email* o *X-Forwarded-User*, che contengono informazioni sull'identità fornite dall'IdP. Questo consente all'applicazione di identificare l'utente senza dover utilizzare la logica OAuth all'interno. [32] [33]

4.4.2 Limitazioni

Anche se ci sono molti vantaggi nell'utilizzo di OAuth2-Proxy, ci sono alcune limitazioni che devono essere prese in considerazione durante la progettazione. L'alta disponibilità è un primo fattore: poiché il proxy non fornisce configurazioni ad alta affidabilità o funzionalità di failover, la ridondanza deve essere gestita manualmente creando più istanze indipendenti. La gestione delle politiche multi-team presenta ulteriori preoccupazioni. Infatti, il sistema non può applicare controlli di autorizzazione a grana fine all'interno della stessa istanza. Pertanto, se diversi gruppi o organizzazioni hanno politiche di accesso diverse, è necessario configurare istanze di proxy distinte.
La fase di configurazione si presenta come un momento estremamente delicato: la sicurezza del sistema è legata ai valori inseriti (es. URL di callback). Una configurazione errata può causare gravi problemi o introdurre vulnerabilità sfruttabili. Di conseguenza, le applicazioni rimangono esposte su Internet anche dopo l'adozione di OAuth2-Proxy, perché l'accesso potrebbe fallire a causa di difetti nel provider OAuth, nel proxy stesso o nella configurazione errata. Per questo, è indispensabile utilizzare connessioni crittografate (HTTPS) e limitare l'accesso ai soli utenti della rete interna per restringere il campo di attacco.

In conclusione, è importante notare che OAuth2-Proxy è stato concepito principalmente per gestire scenari di Single Sign-On e browser-based. Quindi, la gestione nativa si concentra sull'autenticazione via cookie, ma non si occupa di usi più complessi, come l'accesso tramite token per API in ambienti non web. Di conseguenza, i proxy sono ideali per proteggere applicazioni web aziendali e contesti SSO; tuttavia, non sono così efficaci per architetture distribuite complesse o sistemi che richiedono meccanismi di autenticazione altamente personalizzati.

4.5 Confronto tra OAuth puro e OAuth2-Proxy

Per gestire l'autenticazione e l'autorizzazione, OAuth2 può essere utilizzato in vari modi, ciascuno con i propri attributi e conseguenze. Da un lato vi è l'utilizzo di OAuth2 puro,che consente al client di gestire direttamente il flusso di autorizzazione e la conservazione dei token. Dall'altro si colloca l'impiego di strumenti come OAuth2-Proxy, che fungono da intermediario tra l'utente e l'applicazione, rende l'integrazione del protocollo più semplice senza richiedere modifiche al codice del servizio protetto.
La gestione dei token di accesso e di aggiornamento, che è interamente responsabilità del client, è un elemento essenziale di OAuth2 puro. L'applicazione è responsabile della conservazione sicura delle risorse, della gestione corretta delle scadenze e dell'uso appropriato per rinnovare l'accesso alle risorse. Se questo consente al cliente di avere un controllo completo sul processo, comporta problemi di sicurezza. Infatti, una protezione inadeguata dei token può mettere il sistema in pericolo come il furto o l'uso improprio dei token, il che potrebbe compromettere l'integrità dell'intero processo di autorizzazione.

4.5.1 Vantaggi di OAuth2-Proxy rispetto a OAuth2 puro

L'adozione di un OAuth2-Proxy, piuttosto che la gestione diretta del flusso OAuth2 da parte dell'applicazione, offre grandi vantaggi in termini di semplicità, sicurezza e standardizzazione. Ciò è particolarmente vantaggioso in contesti con molte applicazioni.
Uno dei principali vantaggi riguarda la riduzione della complessità applicativa. Le applicazioni non devono implementare direttamente la logica OAuth2 grazie all'OAuth2-Proxy, che consente agli utenti non esperti di proteggere servizi web senza conoscenze approfondite di sicurezza o sviluppo. L'integrazione è più semplice: le applicazioni possono essere posizionate dietro il proxy senza modificare il codice, ricevendo protezione e autenticazione standard immediatamente.
Il proxy, inoltre, è responsabile della sicurezza e della gestione dei token. Mentre l'applicazione riceve solo accesso a token temporanei, i refresh token vengono

mantenuti internamente e non vengono mai mostrati al client. Ciò riduce significativamente il rischio di compromissione dei token a lunga durata: Anche in caso di attacco all'applicazione o a uno dei suoi componenti, l'effetto sarà limitato. Il proxy funge anche da livello di isolamento che impedisce all'applicazione di trattare direttamente i dati sensibili dell'utente, riducendo la superficie di attacco e alleggerendo la responsabilità del cliente nella gestione dei segreti.
Oltretutto, l'OAuth2-Proxy consente di centralizzare la logica di autenticazione e autorizzazione, rendendolo un luogo unico in cui applicare le policy di accesso, gestire la revoca dei permessi e uniformare il logging, la sicurezza e la gestione dei cookie su tutte le applicazioni protette. Questo metodo è particolarmente vantaggioso nei contesti multi-servizio o basati su microservizi, poiché rende l'amministrazione della sicurezza a livello infrastrutturale più semplice e riduce la probabilità di errori o configurazioni errate nell'applicazione. [34]
Infine, il proxy è perfetto per ambienti multi-app o applicazioni legacy che non supportano nativamente OAuth2. Consente un accesso sicuro e standardizzato senza dover riscrivere o aggiornare tutte le applicazioni. Gli utenti beneficiano di un accesso self-service tramite browser senza necessità di installare programmi aggiuntivi, rendendo ancora più semplice utilizzare i servizi.

4.5.2 Svantaggi di OAuth2-Proxy rispetto a OAuth2 puro

Anche se l'adozione di un OAuth2 Proxy offre grandi vantaggi in termini di semplicità, sicurezza e centralizzazione, comporta alcune limitazioni rispetto alla gestione diretta di OAuth2, in particolare in termini di flessibilità, controllo e complessità architettonica.
Un primo problema è che non è così flessibile, e ha alcune limitazioni funzionali. Poiché l'OAuth2-Proxy non mostra i refresh token nelle applicazioni client, alcune funzionalità avanzate, come il login senza username/password o la gestione personalizzata delle sessioni, sono difficili o impossibili da implementare direttamente tramite il proxy. L'uso del proxy può essere troppo severo rispetto a un'implementazione OAuth2 pura, in situazioni che richiedono un controllo minuzioso sui flussi di autenticazione o personalizzazioni avanzate.
Un secondo aspetto da considerare è la maggiore complessità architettonica. Il proxy può causare una latenza aggiuntiva nelle richieste e creare un punto critico di gestione che, se non ridondato correttamente, potrebbe essere un unico punto di failure.

4.6 Integrazione con provider esterni

OAuth 2.0 è attualmente uno degli standard prevalenti per l'autenticazione e l'autorizzazione in applicazioni web e mobili. La sua vasta diffusione è stata guidata

dalla necessità di offrire agli utenti una modalità di accesso più efficace e significativamente più sicura, eliminando la richiesta ripetuta delle credenziali personali per ogni servizio. In questa architettura, la Relying Party (RP) trasferisce la responsabilità dell'autenticazione utente a un Identity Provider (IdP) fidato, come Google o Facebook, con cui l'utente ha già un account. [35]
L'utente, chiamato anche Resource Owner (RO), si autentica direttamente con l'IdP per verificare la propria identità. Dopo l'autenticazione, l'IdP fornisce all'applicazione RP alcuni dati base del profilo (come nome, email o foto), in modo che essa possa creare o associare un utente senza gestire le credenziali. Affinché questo avvenga, la RP deve essere "registrata" in anticipo con l'IdP e avere un client_id, che identifica in modo univoco la RP durante le richieste, e in alcuni casi un client_secret, che serve per effettuare l'autenticazione tra le due parti.
In questo modo, oltre a semplificare la vita all'utente che non deve gestire troppi set di credenziali, OAuth 2.0 incrementa la sicurezza dell'intero meccanismo evitando che le applicazioni debbano maneggiare direttamente password o dati sensibili. [35] [36]

4.6.1 Authorization Code Flow

L'uso di OAuth 2.0 richiede che la Relying Party (RP) completi la registrazione con l'Identity Provider (IdP) per ottenere un client_id univoco. Se la RP rientra nella categoria dei client confidenziali, l'IdP le fornisce anche un client_secret. Questo segreto è essenziale per l'autenticazione crittografica delle chiamate API verso l'IdP stesso.
Il flusso ha inizio quando il Resource Owner (RO) interagisce con l'applicazione, ad esempio cliccando su un pulsante di login. A questo punto la RP costruisce e invia una richiesta all'authorization endpoint dell'IdP. Tale richiesta deve contenere alcuni parametri obbligatori:

- response_type=code: indica che la RP desidera ricevere un authorization code.
- scope: elenca le autorizzazioni specifiche necessarie all'applicazione (es. accesso al profilo base).
- redirect_uri: specifica l'indirizzo di ritorno dell'applicazione, dove l'IdP reindirizzerà l'utente dopo il login.

Il protocollo richiede una validazione rigorosa del redirect_uri da parte dell'IdP per assicurare l'esatta corrispondenza con l'endpoint di callback registrato. Tale meccanismo protegge il sistema da attacchi di reindirizzamento, fungendo da garanzia che l'intero flusso di autenticazione termini sempre in un ambiente fidato.
Dopo aver ricevuto la richiesta dalla Relying Party, l'Identity Provider reindirizza

il Resource Owner verso una pagina di autenticazione ospitata sul proprio dominio. In questa fase l'utente deve effettuare il login con le proprie credenziali presso l'IdP, se non è già in una sessione attiva, nel qual caso anziché richiedere nuovamente il login, l'IdP si limita a mostrare una schermata di consenso in cui l'utente deve accettare la condivisione delle informazioni richieste dalla RP.Se il Resource Owner accorda il consenso l'Identity Provider emette un authorization code, che rappresenta un'autorizzazione temporanea e limitata per poter accedere, in seguito, alle risorse indicate nello scope. Questo codice viene trasmesso dall'IdP alla RP mediante un redirect verso il redirect endpoint, registrato in precedenza e che l'IdP ha verificato. Attraverso questo passaggio infatti il codice viene consegnato solamente all'applicazione legittima e non può essere rubato o utilizzato da terzi non autorizzati.
Una volta ottenuto l'authorization code, la Relying Party effettua lo scambio dei token tramite la propria componente server-side. In particolare l'applicazione effettua una richiesta all'endpoint di token dell'Identity Provider passando sia il codice di autorizzazione ottenuto in precedenza sia il proprio client_secret, che è la credenziale con cui si dimostra che la richiesta è legittima.L'Identity Provider a fronte della validità dell'authorization code e della regolarità della RP (dimostrata attraverso il client_secret) emette un access token. Quest'ultimo rappresenta la credenziale con cui la Relying Party può accedere alle risorse del Resource Owner ospitate presso l'IdP. In questo modo l'applicazione ottiene i permessi necessari senza mai conoscere le credenziali dell'utente e quindi in modo sicuro e indirettamente.
L'intero processo descritto, che costituisce il cuore dell'Authorization Code Flow, è rappresentato in modo schematico nella Figura 4.2, la quale illustra le interazioni tra Resource Owner, Relying Party e Identity Provider.

4.6.2 Analisi di API degli Identity Providers

Un'analisi condotta nel 2021 da Srivathsan G. Morkonda, Paul C. van Oorschot e Sonia Chiasson [36] ha preso in esame gli attributi di dati personali messi a disposizione dai principali provider di identità attraverso i meccanismi basati su OAuth, con l'obiettivo di valutarne le implicazioni in termini di privacy e sicurezza. Gli autori hanno raggruppato tali attributi in cinque categorie distinte: basic, identity, personal, behavioural e sensitive.

La categoria **basic** comprende gli attributi "meno" invasivi in termini di privacy, come il nome, l'email, la foto del profilo. A proposito di nome, addirittura nel Single Sign On di Apple all'utente viene data la possibilità di fornire al relying party un nome a sua scelta, in questo modo a lui viene dato un maggior controllo sui dati forniti.
Gli attributi che fanno parte della categoria **identity** sono più delicati in quanto

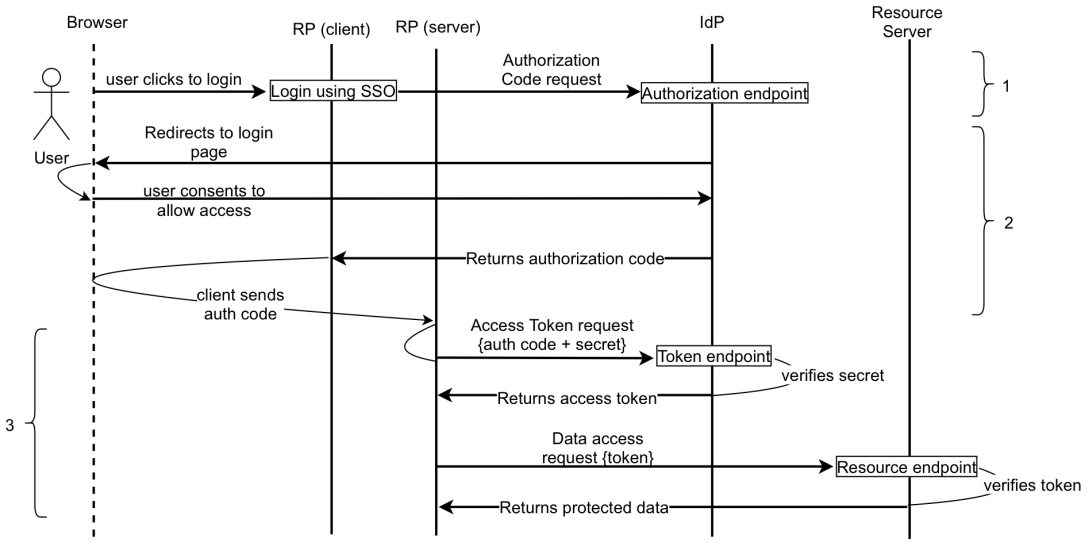

Figura 4.2: Processo per OAuth2.0: Authentication Code Flow

quelli che consentono di identificare gli utenti in modo univoco e quindi quelli che riguardano la data di nascita, il sesso, il numero telefonico, l'indirizzo di casa ecc.... Sono dati delicati in termini di sicurezza e che se finissero nelle mani di un attaccante potrebbero essere usati per attività fraudolente come ad esempio chiedere un nuovo numero di telefono o una nuova sim.
La categoria **personal** riguarda gli attributi che danno accesso ai dati personali di un utente, come foto, video o la posizione geografica. A essa si affianca la categoria **behavioural** che includerebbe i dati che possono rivelare abitudini, gusti e personalità di un utente, come i social a cui l'utente ha dato un "like" o le interazioni di cui l'utente è stato protagonista ecc.... Questo tipo di dati può essere di interesse per terze parti come traccianti o script di advertising che sono interessati a profilare il comportamento degli utenti.
Infine la categoria **sensitive** a cui appartengono i dati più sensibili come la rubrica, le email, i file ecc.... Sono spesso dati che contengono informazioni di estrema riservatezza e l'accesso a questi dati è pericoloso perchè potrebbe ledere direttamente la sfera privata e lavorativa dell'utente

Google OAuth API

La piattaforma OAuth 2.0 di Google organizza i dati degli attributi degli utenti in base alle API dei vari servizi che fanno parte del suo ecosistema. Ad esempio, l'API di Gmail raggruppa gli attributi rilevanti per l'invio e la ricezione di e-mail in una casella di posta Gmail. Questo è utile per i client di posta elettronica di terze parti. Le informazioni personali dell'utente come data di nascita, sesso e informazioni

Data category	Google	Facebook	Apple	LinkedIn
Basic	email *profile* openid	email *public_profile*	email name	r_emailaddress name profilePicture headline
Identity	user.birthday.read user.addresses.read* user.gender.read* user.phonenumbers.read*	user_birthday user_hometown user_gender user_age_range instagram_graph_user_profile*		address birthDate phoneNumbers backgroundPicture
Personal	userinfo.profile photoslibrary* fitness* tasks*	user_location user_photos user_videos instagram_graph_user_media*		geoLocation
Behavioural	games* user.organization.read*	user_likes user_posts user_link		organizations positions educations projects certifications skills volunteeringInterests volunteeringExperiences
Sensitive	contacts drive gmail documents* spreadsheets* youtube*	user_friends		websites industryName courses testScores summary

Figura 4.3: Confronto degli attributi dei dati nelle api dei fornitori. *dati non richiesti (ma disponibili) da nessun sito web nel nostro set di dati. i campi predefiniti sono in corsivo.

di contatto sono esposte tramite People API. Altre API di Google comunemente utilizzate dagli RP includono l'API del calendario e l'API Drive (utilizzata per accedere allo spazio di archiviazione di Google Drive dell'utente).
Un altro aspetto che caratterizza la piattaforma è che alcune proprietà vengono classificate come sensibili o ristrette. In questo caso i Client che vogliono accedervi devono farsi verificare formalmente da Google per dimostrare che ne hanno diritto e che li proteggono come si deve.
Per quanto riguarda le informazioni sul profilo, la piattaforma distingue due principali. Da un lato abbiamo il profile, che fa parte del basic, che contiene un numero molto ridotto di informazioni, per esempio nome, email e foto profilo e che viene fornito di default in tutte le risposte di autenticazione. Dall'altro abbiamo lo userinfo.profile che contiene un insieme più ampio di informazioni disponibili pubblicamente nel profilo dell'utente, ma che deve essere richiesto dal RP.
Google OAuth 2.0 supporta inoltre l'attributo openid, che consente agli RP di identificare univocamente gli utenti come specificato dalla specifica OpenID Connect 1.0. In questo modo si consente all'RP di ottenere da Google un token ID contenente campi che confermano l'identità dell'utente (ad esempio, nome utente,

emittente del token, timestamp di scadenza del token). I campi sono codificati come coppie chiave-valore nella porzione di dati di un oggetto JWT (JSON Web Token) con una firma digitale firmata da Google. [36]

Facebook OAuth API

Per quanto riguarda Facebook (Figura 4.4), l'API che si occupa dell'autenticazione

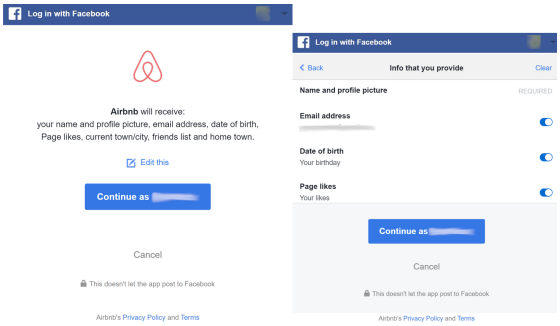

Figura 4.4: Facebook SSO [36]

tramite OAuth 2.0 è Graph API, che interagisce con i dati di Facebook di un utente. In questo modo, i responsabili possono accedere in lettura a una varietà di dati dal profilo Facebook dell'utente. Essendo un social media, il login tramite Facebook fornisce dati comportamentali specifici, come like delle pagine Facebook o post social create dall'utente che non sono disponibili tramite altri IdP. Oltre alle caratteristiche di base di Facebook, altre autorizzazioni sono molto utili per la gestione aziendale, come consentire alle applicazioni di visualizzare, creare, modificare ed eliminare contenuti sulle pagine Facebook amministrate dall'utente e utilizzarle per promuovere prodotti e servizi.
Tuttavia, è importante tenere presente che gli RP che richiedono autorizzazioni diverse da public_profile (nome dell'utente e immagine del profilo), e-mail e pages_show_list (elenco delle pagine Facebook gestite dall'utente) devono essere sottoposti a una revisione dell'app di Facebook per valutare la legittimità della richiesta in conformità con le proprie regole.
Una considerazione significativa è che, sebbene Facebook Login non supporti direttamente la specifica OpenID Connect (OIDC), consente alle parti affidabili di autenticare gli utenti senza necessariamente accedere ai loro dati personali. In realtà, l'autenticazione avviene tramite una richiesta firmata, che è un oggetto JSON codificato in base64url e firmato digitalmente che contiene un identificativo utente rilasciato direttamente da Facebook.[36]

Apple OAuth API

Nel 2019 Apple ha introdotto iI framework **Sign in with Apple** (Figura 4.5),

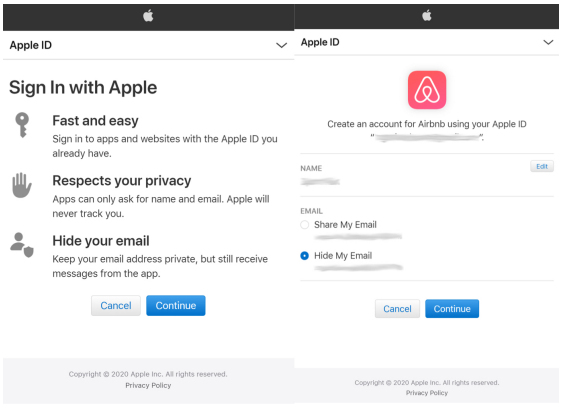

Figura 4.5: Apple SSO [36]

concepito come alternativa rispettosa della privacy per gli utenti che desiderano accedere a applicazioni web e mobili di terze parti tramite Single Sign-On (SSO) senza divulgare i propri dati personali. L'accesso con Apple, infatti, consente ai relying party (RP) di autenticare gli utenti e richiedere facoltativamente l'accesso a informazioni private come il nome e l'indirizzo email, attraverso l'uso del parametro scope.
Quando richiesto, Accedi con Apple offre una funzionalità distintiva, che lo distingue rispetto alle metodologie offerte dagli altri competitor. Infatti è possibile scegliere se condividere il proprio indirizzo email reale oppure usare un indirizzo completamente anonimo generato dal servizio Private Email Relay di Apple. In questo modo viene generato un indirizzo e-mail anonimo unico per la coppia utente-RP e indirizza tutta la corrispondenza e-mail tra RP e utente attraverso questa e-mail, senza rivelare l'indirizzo effettivo dell'utente al servizio richiedente, garantendo un livello superiore di protezione della privacy, impedendo quindi di profilare o tracciare gli utenti sulla base della loro email. Inoltre, le applicazioni che si trovano sull'App Store di Apple e utilizzano un servizio SSO di terze parti (come Facebook e Google) sono ora tenute a integrare e offrire lo stesso servizio come opzione SSO da parte di Apple.
Infine, il framework incorpora un meccanismo utile ai RP che consente di distinguere tra utenti reali e entità automatizzate. Si tratta del valore booleano fornito durante l'autenticazione, noto come indicatore reale dell'utente. Questo aumenta l'affidabilità dei processi di accesso e riduce il rischio di abusi legati a bot o identità sintetiche. [36]

4.7 Rischi di sicurezza comuni e mitigazioni

Implementato senza le best practice evidenziate ad esempio da OWASP, da Google, da Auth0 o da Okta, OAuth 2.0 può rappresentare un pericolo per le applicazioni e per gli utenti abilitati. Il problema è da ricercare sia nel lavoro di integrazione delicato e articolato, sia nelle impostazioni o sottoimpostazioni a volte non corrette che possono allentare le difese.
Le minacce a Auth 2.0 oggi rilevanti possono essere individuate nelle liste pubbliche di vulnerabilità e debolezze delle applicazioni come CVE e CWE.
I CVE (Common Vulnerabilities and Exposures) sono "elencazioni" pubbliche di vulnerabilità di sicurezza riferite a software e sistemi identificate da un nome chiaro e univoco ed un breve abstract, e che spesso rimandano ad una capitalization. I CWE (Common Weakness Enumeration) sono categorie, più o meno articolate e gerarchicamente ordinate, di vulnerabilità più o meno generiche nella codifica o nella progettazione del software, che possono condurre a vulnerabilità di sicurezza. Sia i CVE che i CWE aiutano a categorizzare le vulnerabilità legate all'ecosistema Auth 2.0, e quindi a esplorare gli attacchi e a identificare le contromisure corrette. [37]

4.7.1 Phishing tramite Redirection URI (Open Redirect)

L'utilizzo improprio del parametro *redirect_uri* è una grave falla di OAuth 2.0 che può essere utilizzata in attacchi di tipo open redirect. Poiché questo tipo di attacco sfrutta la mancanza di validazione del server di autorizzazione, che espone le credenziali o i token di accesso dell'utente, è possibile che un attaccante reindirizzi l'utente a un sito malevolo sotto il suo controllo.
Questo attacco è basato infatti su una mancata o insufficiente validazione dell'URL di redirezione, che permette all'attaccante di manipolare il percorso (path confusion) o i parametri (parameter pollution) dell'URL. In questo modo, la risposta di autenticazione viene dirottata verso un dominio non autorizzato. Ciò consente all'attaccante di intercettare direttamente il codice di autorizzazione o i token OAuth 2.0. In situazioni più complesse, questa tecnica può essere usata insieme ad altre falle di sicurezza web, portando a un attacco che compromette l'intero processo di autenticazione e annulla le garanzie del protocollo.
Di conseguenza, è essenziale una validazione approfondita del *redirect_uri*, assicurando la sua coincidenza esatta con uno degli *endpoints* memorizzati. Un'ulteriore precauzione fondamentale è garantire che la destinazione finale del reindirizzamento non possa essere modificata da parametri aggiuntivi o da manipolazioni del percorso. [37] [38]

4.7.2 Token Leakage (Implicit Flow)

Le applicazioni client-side come le Single Page Applications (SPA) hanno storicamente utilizzato il flusso implicito di OAuth2 perché consentiva di ottenere rapidamente un access token senza dover passare attraverso un backend. Tuttavia, questo metodo è oggi considerato insicuro e deprecato, principalmente a causa del fenomeno noto come "token leakage", in cui terze parti non autorizzate possono accidentalmente accedere ai token.
Nell'Implicit Flow, l'accesso al token viene restituito direttamente nell'URL di riferimento, solitamente rappresentato da un *fragment*. Ciò lo rende vulnerabile a numerose intercettazioni che possono essere effettuate tramite log del browser, referrer che vengono inviati a domini terzi, cronologia e script malevoli installati dal client. L'assenza del backend aggrava la situazione. In effetti, non c'è un backend che possa salvaguardare o gestire la sicurezza del token poiché viene consegnato direttamente al client (generalmente una SPA o un'app mobile).
Le linee guida più recenti suggeriscono di sostituire l'Implicit Flow con un flusso di autorizzazione arricchito dal meccanismo PKCE (Proof Key for Code Exchange), che non mostra i token direttamente nell'URL e riduce significativamente le possibilità di intercettazione. Inoltre, possono essere prese misure per legami di token come il DPoP (Demonstration of Proof-of-Possession), che impone l'utilizzo del token a clienti legittimi, e l'uso obbligatorio di connessioni HTTPS crittografate, che è essenziale per prevenire attacchi di tipo MITM e garantire la riservatezza della trasmissione delle credenziali. [39] [40]

4.7.3 CSRF durante il login OAuth

Il rischio di attacchi **CSRF** (Cross-Site Request Forgery) durante la fase di autenticazione aumenta in particolare quando il parametro *state*, progettato per proteggere l'integrità delle richieste, non viene implementato o validato correttamente. In effetti, il parametro *state* può collegare la richiesta di autenticazione a una particolare sessione utente. Ciò impedisce alle richieste irregolari di essere accettate come valide.
Questa tipologia di attacco sfrutta l'assenza di questo parametro o la sua gestione errata. Un attaccante può creare un link malevolo che avvia una richiesta OAuth apparentemente valida, ma associata al proprio *client ID* se lo *state* non viene generato in modo sicuro, è assente o statico. L'utente vittima viene indotto a registrarsi in questo modo, consentendo all'attaccante di accedere alle sue risorse protette. In pratica, il server di autorizzazione non è più in grado di determinare se la richiesta provenga effettivamente da un cliente legittimo o da un attore malvolo. Recenti ricerche rivelano che oltre un quarto, se non più di un terzo, dei siti testati, che utilizzano OAuth, sono ancora vulnerabili a questo tipo di attacco. Gli errori

più frequenti alla base di ciò vanno dalla semplice disattenzione alla mancata conoscenza dell'importanza del parametro state. La potenziale posta in gioco per questo attacco è alta: un finto attaccante che ne approfitta può letteralmente ottenere le chiavi di accesso a tutte le risorse dell'utente e dunque aggirare le difese che il protocollo OAuth2 dovrebbe garantire. La comunità di sicurezza e le best practice propongono un metodo chiaro per ridurre il rischio: Il parametro state deve essere sempre generato in modo sicuro, distinto e imprevedibile, inviato all'authorization server e, soprattutto, deve essere rigorosamente validato al ritorno della risposta. Qualsiasi discrepanza tra il valore inviato e quello ricevuto deve essere rifiutata immediatamente. È possibile ridurre significativamente l'esposizione al rischio di CSRF nel contesto di OAuth2 trattando state come un requisito obbligatorio piuttosto che come una semplice opzione. [41] [35]

4.7.4 Refresh token theft

Il furto del refresh token rappresenta una delle minacce più rilevanti per l'ecosistema OAuth2, perchè consente ad un attaccante di ottenere l'accesso alle risorse anche dopo la scadenza del normale access token. Infatti, a differenza di quest'ultimo, il refresh token consente di richiedere nuovi access token senza che l'utente debba autenticarsi nuovamente, garantendo la continuità della sessione. Emerge quindi, come la sua violazione costituisca un pericolo significativo per la sicurezza del protocollo.
Il funzionamento è semplice, ma devastante e il suo impatto può essere anche amplificato in altri scenari come quello dei Family Refresh Tokens (FRTs), ovvero token il cui accesso può essere esteso a più client, caratterizzando una compromissione sistemica, che favorisce forme di escalation dei privilegi e amplifica la superficie dell'attacco
In ambienti come quelli legati alle applicazioni web, ai browser e ai dispositivi mobili, il rischio è più alto. In questi scenari infatti i refresh token possono esposti se non sono protetti adeguatamente. Il loro utilizzo in contesti client-side, ad esempio, rende più probabile che vengano intercettati o sottratti tramite malware, script malevoli o attacchi man-in-the-middle, soprattutto in assenza di connessioni sicure.
Per mitigare questa minaccia, sono state sviluppate una serie di strategie, che si fondano su un unico principio: i refresh token devono essere trattati come credenziali di massima sensibilità. Infatti, la loro conservazione deve avvenire esclusivamente in ambienti sicuri, preferibilmente lato server, evitando l'archiviazione nel frontend o all'interno di browser potenzialmente compromessi. Un approccio consolidato in questo senso è il modello Backend for Frontend (BFF), che prevede la gestione dei token unicamente dal backend, riducendo l'esposizione sul lato client.
Altre tecniche di difesa consistono nel binding del token a parametri specifici,

come quello della sessione o del dispositivo utilizzato, come nel caso del browser fingerprinting. Questo si pone di impedire l'utilizzo del token in contesti diversi da quelli legittimi. La rotazione periodica dei refresh dei token riduce i rischi riducendo la finestra temporale di esposizione in caso di furto e consentendo la revoca immediata in caso di comportamenti sospetti.
Infine, la multi-factor authentication (MFA) aggiunge un ulteriore livello di protezione. In questo modo, anche nel caso in cui un attaccante riesca a ottenere un token di ricarica, l'assenza di ulteriori fattori di autenticazione gli impedirebbe di sfruttarlo.

Capitolo 5

Analisi dell'applicazione PWA

5.1 Descrizione della PWA

L'applicazione sviluppata rappresenta una Progressive Web App dimostrativa che simula un sistema di prenotazione con funzionalità di autenticazione moderna. Non si tratta quindi di un sistema già esistente preso di mira per la ricerca, ma di un'applicazione ad hoc sviluppata in un ambiente non produttivo come caso di studio per verificare come integrare tecnologie contemporanee e moderne di autenticazione e concentrarsi su come implementare un accesso sicuro e intuitivo alle risorse protette.
In questa prospettiva, l'applicazione è stata volutamente ridotta alle sole funzionalità essenziali. Dopo il login, l'utente viene reindirizzato a un'area riservata semplice. L'applicazione comprende inoltre un menu che indica lo stato di autenticazione, indicatori di sessione attiva e funzionalità di logout per poter gestire l'eventuale chiusura esplicita della sessione.
Tutto è stato implementato con precise limitazioni. Non ci sono logiche di business complesse: si utilizzano soltanto configurazioni pensate per l'ambiente di sviluppo e non di produzione, si usa un service worker molto semplice e un database in memoria con dati fittizi. Ciò ha permesso di mantenere l'attenzione sugli scopi della sperimentazione e della dimostrazione della possibile integrazione dei due standard di autenticazione in una moderna PWA.
Tutte le interfacce sono state realizzate applicando i principi del Material Design per garantire un'esperienza utente moderna e intuitiva, ottimizzata per tutti i dispositivi e le risoluzioni. Di seguito sono riportate le varie interfacce.

5.1.1 Schermata principale e navigazione

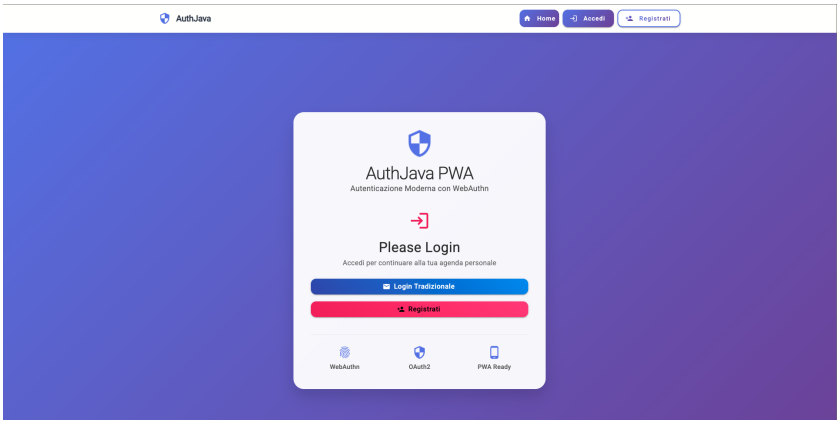

Figura 5.1: Schermata Home

La home page dell'applicazione offre un'interfaccia gradevole (Fig. 5.1), che introduce l'utente al sistema. Le funzionalità principali dell'app sono accessibili dalla home page, che guida l'utente verso le operazioni chiave tramite un menu di navigazione. La barra di navigazione in alto, invece, permette di raggiungere velocemente le varie sezioni dell'app, a seconda che l'utente sia connesso o meno. Infatti, se l'utente non è loggato, il menu mostra le opzioni per fare login o iscriversi. Dopo aver completato l'autenticazione, la navigazione si arricchisce poi di collegamenti tra cui il proprio calendario e la propria dashboard. Cambiando dinamicamente il layout viene fornito un feedback immediato e chiaro sullo stato di autenticazione.

5.1.2 Processo di registrazione

Se l'utente è nuovo al sistema, può decidere di registrarsi tramite la pagina apposita (Fig. 5.2). Questa è una semplice schermata per la raccolta dei dati. L'interfaccia è molto pulita e propone un form essenziale e minimale in cui inserire i dati, quali username, email e password.

5.1.3 Interfaccia di autenticazione

La schermata di login (Fig. 5.3) è la pagina più importante dell'applicazione. La sua interfaccia pulita e professionale riflette l'approccio moderno adottato per la gestione dell'autenticazione, con un design intuitivo e coerente con tutte le altre schermate. Ciò facilita il processo di accesso tramite metodi di autenticazione sicuri e consolidati. La caratteristica particolare di questa pagina è che si può accedere

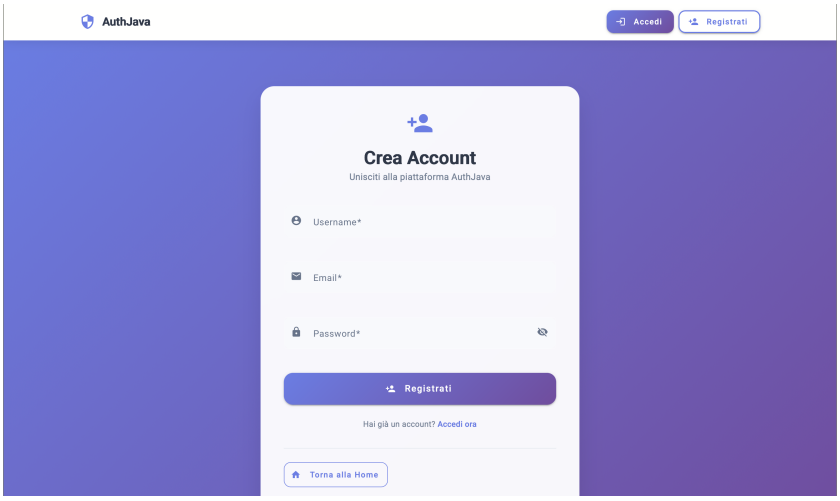

Figura 5.2: Schermata di registrazione

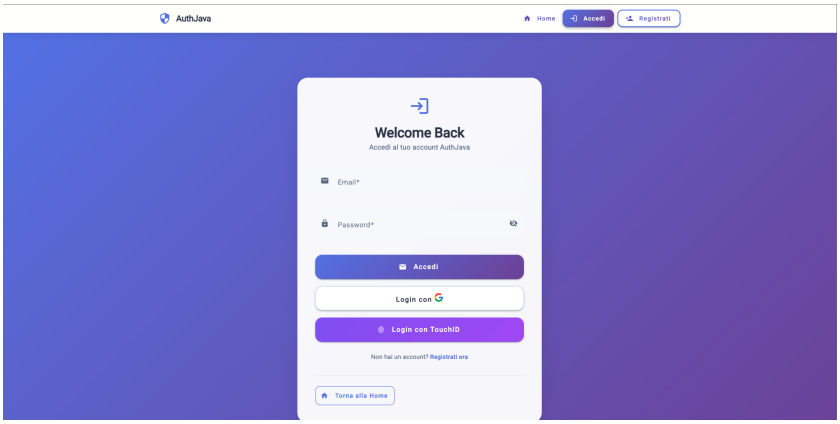

Figura 5.3: Schermata di login, con possibilità di effettuare l'accesso con Google

tramite Google, cioè gli utenti possono usare le proprie credenziali di Google per fare il login e usufruire dell'infrastruttura e delle funzionalità di sicurezza avanzate fornite da Google in qualità di Identity Provider. Anche in questo caso, l'interfaccia fornisce feedback visivi chiari durante tutto il processo di autenticazione.

5.1.4 Dashboard e gestione del profilo

Dopo il login, l'utente viene indirizzato al proprio profilo (Fig. 5.4). In questa pagina sono raccolte tutte le informazioni necessarie riguardanti l'autenticazione. Da questa pagina e da altre è già possibile effettuare il logout sicuro che disattiva tutte le credenziali di autenticazione attive e reindirizza l'utente all'area pubblica

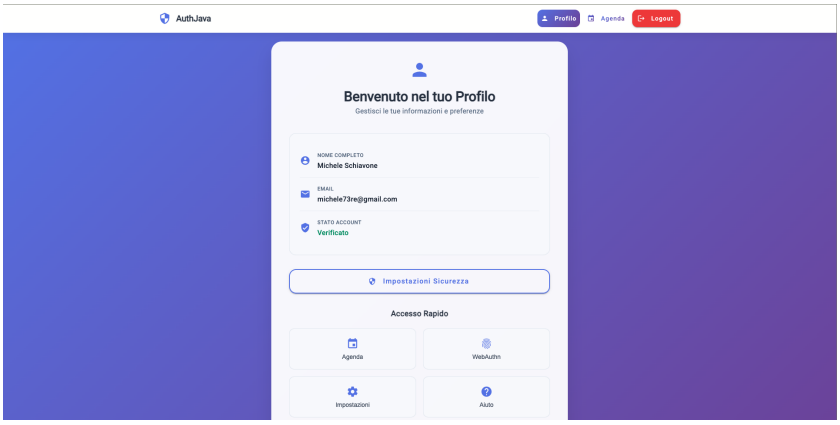

Figura 5.4: Area protetta dell'utente

dell'applicazione. L'intero procedimento di logout assicura la totale eliminazione dello stato di accesso, in modo che non restino tracce delicate archiviate nel browser o nel sistema.

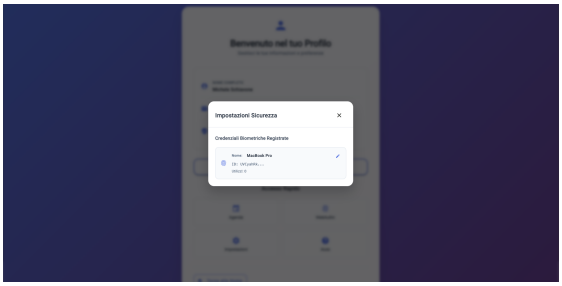

Figura 5.5: Possibilità di visualizzare le credenziali registrate e rinominarle

Oltre a mostrare i dati personali, la schermata principale ha link che portano l'utente alle funzionalità principali dell'app, come l'agenda, o che gli consentono di gestire le proprie impostazioni relative alla protezione. Questi elementi contribuiscono a rendere trasparente il funzionamento delle diverse autenticazioni attive e permettono all'utente di avere il pieno controllo della propria esperienza con l'applicazione. Tra queste, è possibile trovare le impostazioni di sicurezza (Fig. 5.5) che permettono di visualizzare e rinominare le credenziali registrate per riconoscere subito i dispositivi, le impostazioni per eliminare l'account dalla piattaforma (Fig. 5.6) e verificare che le credenziali WebAuthn siano state implementate correttamente (Fig. 5.7). Infine, è possibile ottenere assistenza sulle funzionalità principali dell'applicazione (Fig. 5.8) . Quest'ultimo è solo un pulsante informativo, non utile per un'applicazione già costituita per la produzione.

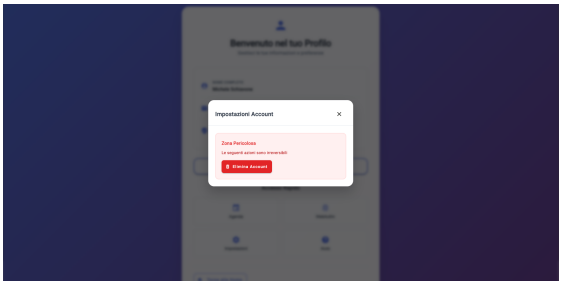

Figura 5.6: Impostazioni che permettono di eliminare il proprio account

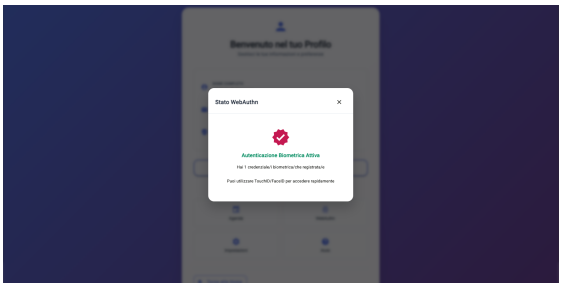

Figura 5.7: Controllo dell'avvenuta registrazione delle credenziali

5.1.5 Gestione degli appuntamenti

A scopo dimostrativo, è stato sviluppato un sistema di gestione degli appuntamenti su cui si basa la schermata dell'agenda (Fig. 5.9) Qui è possibile visualizzare e gestire gli appuntamenti associati al profilo dell'utente autenticato. L'interfaccia dell'agenda presenta un layout semplice e ordinato che consente di scorrere le informazioni.
L'agenda dimostra come i dati personali siano effettivamente associati all'identità dell'utente autenticato. La presentazione delle informazioni segue standard di usabilità consolidati, garantendo un'esperienza intuitiva anche per gli utenti meno esperti di strumenti digitali.

5.1.6 Esperienza utente responsive

L'applicazione sviluppata ha naturalmente un'altra importante qualità, che necessita di essere menzionata. Si tratta della esperienza utente responsive, caratteristica delle Progressive Web App. In effetti, essa si adatta alle caratteristiche del dispositivo utilizzato automaticamente. Questo può essere un computer desktop, un tablet o uno smartphone (Fig 5.10). Di conseguenza, tutte le funzionalità principali possono essere facilmente accessibili e utilizzate.
Ciò mostra come le tecnologie web possono avvicinarsi a un'esperienza di tipo nativo

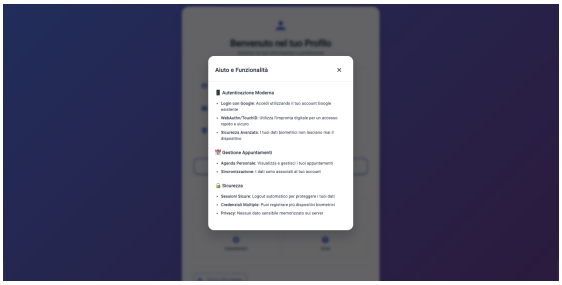

Figura 5.8: Aiuto e funzionalità

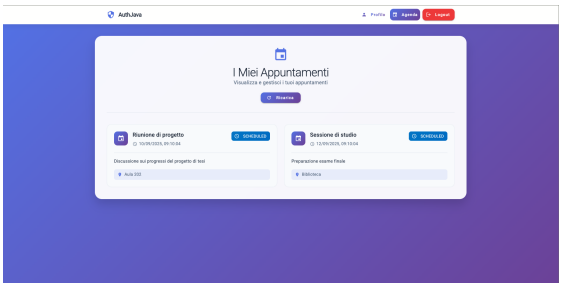

Figura 5.9: Agenda personale

mantenendo la facilità tipica di distribuzione e aggiornamento delle applicazioni web. Con controlli ottimizzati per l'uso touch sui dispositivi mobili e per l'uso mouse e tastiera sui dispositivi desktop, la navigazione viene gestita in modo intelligente in base ai diversi modi in cui l'utente interagisce. In questo modo, l'applicazione risulta sempre utilizzabile ed efficiente in tutti i casi d'uso previsti.

5.2 Stack tecnologico

La PWA utilizza una combinazione di tecnologie moderne sia front-end che back-end per garantire un equilibrio tra facilità di implementazione, rapidità di sviluppo e capacità di validare scenari di autenticazione avanzata.
Per il frontend è stato adottato Angular 17, un framework che rappresenta uno standard consolidato per la creazione di applicazioni a una pagina. Con il supporto nativo per PWA, è perfetto per testare meccanismi di autenticazione in ambienti client-side. Angular Material è stato utilizzato per garantire la coerenza grafica e ridurre i tempi di sviluppo grazie a componenti predefiniti e facilmente integrabili.
Al contrario, il backend è stato sviluppato con Spring Boot 3, che fornisce un ambiente di sviluppo flessibile, scalabile e configurabile in un batter d'occhio. Per quanto riguarda la persistenza dei dati, è stato utilizzato H2 in-memory, un database che, sebbene non sia stato progettato per contesti di produzione, è adatto a

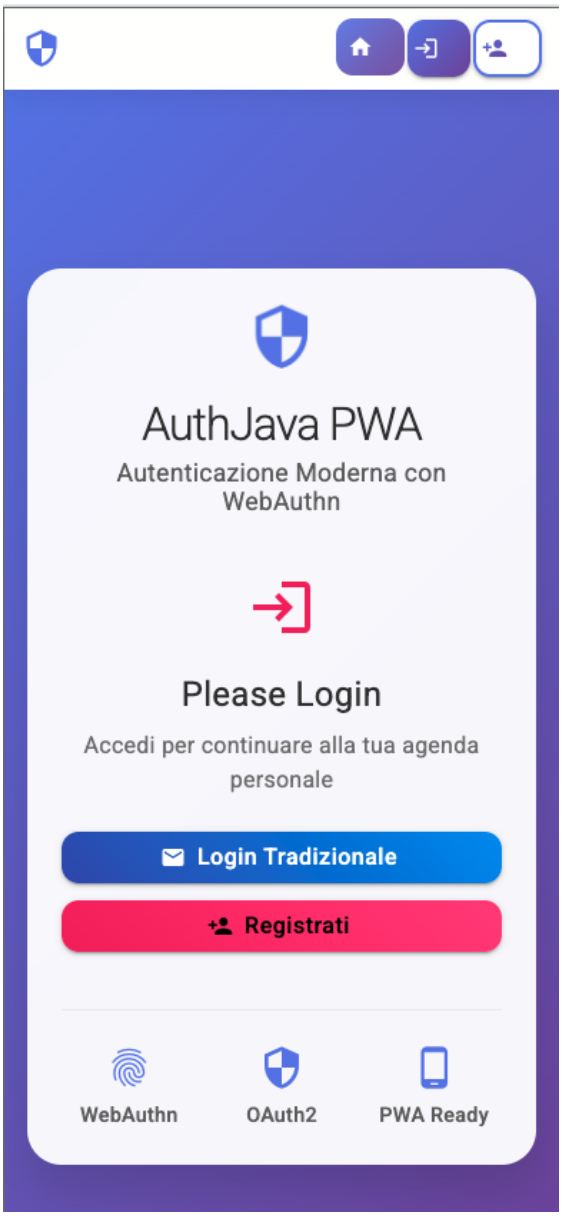

Figura 5.10: Visualizzazione della PWA su un iPhone 12 Pro

questo scenario volutamente prototipale, consentendo cicli di test rapidi e rendendo più semplice gestire le informazioni legate agli utenti e i dati ad essi correlati.
Per raggiungere gli obiettivi di questa tesi e integrare gli standard di autenticazione precedentemente citati, è centrale nell'architettura l'utilizzo di WebAuthn4J, una libreria lato server dedicata alla verifica delle operazioni di registrazione e autenticazione basate su WebAuthn. Per garantire la massima portabilità tra i vari

framework, si concentra esclusivamente sulla verifica lato server delle attestazioni e delle autenticazioni. Infatti, questa libreria contiene limitazioni intenzionali: il suo scopo è limitato alla validazione crittografica delle risposte inviate dal client e non include funzionalità relative alla gestione del flusso applicativo completo di OAuth2. Alcune funzionalità, come l'eliminazione dei parametri dalle richieste HTTP, la gestione e la persistenza delle sfide, il salvataggio delle chiavi pubbliche come credenziali utente o il loro recupero in fase di autenticazione, devono essere implementate direttamente dall'applicazione che la integra. Questa caratteristica è un vantaggio perché consente di adattare l'integrazione alle specifiche esigenze architetturali del sistema, ma richiede anche più lavoro di sviluppo.
Dunque, la configurazione è stata deliberatamente mantenuta al minimo dal punto di vista PWA: l'obiettivo principale rimane la validazione dell'autenticazione senza password, quindi non sono stati implementati service worker avanzati né meccanismi di caching complessi per il supporto offline.

5.3 Architettura di base della PWA

Come già detto, l'applicazione segue un modello a più livelli, con una netta separazione tra il frontend e il backend, riflettendo le best practice dello sviluppo web moderno, con una chiara separazione tra livello di presentazione e livello di business. Questo assicura una netta modularità e la possibilità di evolvere in maniera indipendente i vari livelli, aspetto fondamentale nello sviluppo di applicazioni scalabili e mantenibili.
Il frontend è responsabile dell'interfaccia grafica e della gestione dei flussi di autenticazione tradizionali. Dall'altro lato, il backend è pensato per simulare un dominio applicativo e fornire le API necessarie al test, si è infatti pensato ad un caso d'uso di una agenda che deve fornire degli appuntamenti ad un utente loggato nel sistema. Questa separazione consente quindi di isolare la logica di presentazione da quella di business, rendendo l'applicazione più modulare e facilmente estendibile, seguendo i principi di sviluppo software moderni

5.3.1 Livello Frontend: Gestione delle Interfacce e Controllo di Accesso

Sul lato frontend sono stati implementati i componenti essenziali per la dimostrazione.
Dal punto di vista tecnico, meccanismi come gli Auth Guards, mostrati nel Listing 5.1 e 6.1, impediscono agli utenti non autenticati di accedere direttamente alle rotte riservate, simulando così il comportamento normale di un sistema reale.

In questo caso sono stati implementati due controlli doppi che si integrano a vicenda: l'‘authGuard‘ protegge le rotte per gli utenti autenticati, richiedendo come minimo il login, e nel caso in cui non fossimo loggati, ci rimanda a quest'ultima; il ‘guestGuard‘, invece, impedisce l'accesso alle pagine di login per un utente già loggato, che quindi viene reindirizzato automaticamente alla sua dashboard.
Questa architettura di controllo si rivela particolarmente efficace nell'ambito della PWA sviluppata, dove le sezioni dell'agenda e del profilo utente necessitano di protezione, mentre le pagine di registrazione e login devono rimanere accessibili esclusivamente agli utenti non autenticati. Il meccanismo degli Auth Guards dimostra come sia possibile implementare controlli di sicurezza sofisticati nel frontend pur mantenendo la logica di business separata nel backend.

```
export const authGuard: CanActivateFn = (route, state) => {
  const authService = inject(AuthService);
  const router = inject(Router);

  return authService.checkLogin().pipe(
    map((isLoggedIn: boolean) => {
      if (isLoggedIn) {
        return true;
      } else {
        // Se NON loggato, redirect alla pagina di login
        return router.parseUrl('/login');
      }
    })
  );
};
```

Listing 5.1: Auth Guard

```
export const guestGuard: CanActivateFn = (route, state) => {
  const authService = inject(AuthService);
  const router = inject(Router);

  return authService.checkLogin().pipe(
    map((isLoggedIn: boolean) => {
      if (!isLoggedIn) {
        return true; // Se NON loggato => OK
      } else {
        return router.parseUrl('/profile'); // Se già loggato =
   Redirect a /welcome
      }
    })
  );
};
```

Listing 5.2: Guest Guard

5.3.2 Livello Backend: Servizi di Supporto e Gestione dei Dati

Il backend Spring Boot opera come livello di servizio, esponendo API REST che gestiscono il dominio applicativo degli appuntamenti. Sebbene volutamente semplificato per fini dimostrativi, questo livello mantiene una struttura estensibile. L'entità 'Appointment', illustrata nel Listing 5.3, costituisce il nucleo del modello di dominio e dimostra come sia possibile strutturare dati complessi mantenendo la semplicità. L'entità incapsula le informazioni essenziali di un appuntamento, quali titolo, descrizione, data/ora, ubicazione e stato, associandole all'email dell'utente proprietario attraverso un meccanismo di ownership che garantisce l'isolamento dei dati tra utenti diversi.
Il layer di servizio implementa operazioni CRUD complete che coprono l'intero ciclo di vita degli appuntamenti: dalla creazione e lettura fino alla modifica e cancellazione. Particolare attenzione è stata dedicata alle query di ricerca, che permettono il recupero di appuntamenti per utente specifico, per range temporali e per stato, funzionalità che riflettono le esigenze reali di un'applicazione di gestione calendario..

```
1  /**
2   * Entità JPA per la gestione degli appuntamenti nel sistema di
       agenda
3   * Implementa un modello di sicurezza basato su email utente per l
       'isolamento dei dati
4   */
5  @Entity
6  @Table(name = "appointments")
7  public class Appointment {
8
9      // Chiave primaria auto-generata
10     @Id
11     @GeneratedValue(strategy = GenerationType.IDENTITY)
12     private Long id;
13
14     // Campi obbligatori per la definizione base dell'appuntamento
15     @Column(nullable = false)
16     private String title;                    // Titolo dell'
       appuntamento
17
18     @Column(nullable = false)
19     private LocalDateTime dateTime;          // Data e ora dell'
       appuntamento
20
21     // Campi opzionali per dettagli aggiuntivi
22     private String description;              // Descrizione
       dettagliata
```

```
23      private String location;                    // Luogo dell'
        appuntamento
24
25      // Campo di sicurezza per l'isolamento multi-utente
26      @Column(nullable = false)
27      private String userEmail;                   // Email del
        proprietario (chiave di sicurezza)
28
29      // Costruttore completo per la creazione programmatica
30      public Appointment(String title, String description,
        LocalDateTime dateTime,
31                         String location, String userEmail) {
32          this.title = title;
33          this.description = description;
34          this.dateTime = dateTime;
35          this.location = location;
36          this.userEmail = userEmail;
37      }
38
39      // Getter/Setter standard JPA...
40  }
```

Listing 5.3: Entità Appointment

5.4 Requisiti funzionali e non funzionali

L'applicazione è stata progettata principalmente per soddisfare le esigenze funzionali relative alla validazione dei flussi di autenticazione, prestando particolare attenzione agli standard contemporanei come WebAuthn.
Al fine di rendere il prototipo estensibile e comprensibile, le scelte architetturali hanno privilegiato la semplicità e la chiarezza dal punto di vista non funzionale. Per garantire la rapidità dello sviluppo e della riproducibilità degli esperimenti, sono state utilizzate soluzioni leggere e controllate, come l'uso di un database in memoria per i test e una configurazione di sicurezza ridotta al minimo. Inoltre, si è verificato che i flussi WebAuthn funzionassero correttamente su diversi dispositivi e sistemi, come browser come Chrome e Safari, per concentrarsi su questioni come la portabilità e la compatibilità cross-browser.

Capitolo 6

Progettazione e implementazione dell'autenticazione fingerprint

6.1 Obiettivi dell'integrazione

La fase iniziale mirava a implementare l'autenticazione biometrica su dispositivo, delegando la gestione della sicurezza a un IdP esterno e utilizzando una infrastruttura proxy preesistente come OAuth2-Proxy. Questa strategia è stata adottata per evitare la gestione diretta di aspetti complessi (come la validazione dei token e la gestione delle sessioni) così da potersi concentrare al miglioramento dell'esperienza utente e alle funzionalità della Progressive Web App.
Nonostante ciò, questa soluzione iniziale si è dimostrata limitante sotto diversi aspetti. Era evidente la necessità di un'architettura più evoluta, che fosse capace non soltanto di integrarsi in modo ottimale con un IdP esterno per l'identificazione, ma anche di custodire e gestire in modo autonomo i dati biometrici sensibili collegati a ogni specifico account.
Per superare tali limitazioni, è stata progettata e realizzata una soluzione dedicata: un proxy personalizzato in grado di orchestrare l'intero processo di autenticazione, integrarsi con un IdP conforme agli standard (come OAuth 2.0 e OpenID Connect) e gestire in modo sicuro la registrazione e l'associazione delle credenziali biometriche degli utenti.
Il flusso risultante prevede l'utilizzo da parte dell'utente delle credenziali dell'Idp per accedere al sistema e poi associare un'impronta digitale al proprio account.

Da quel momento l'utente può accedere tramite l'impronta digitale memorizzata ottenendo lo stesso livello di accesso garantito dall'Idp, senza dover ridigitare la propria password e i problemi già accennati nel capitolo 2.

6.2 Scelte architetturali

La nuova architettura dell'applicazione mantiene l'approccio multilivello, dividendo chiaramente le responsabilità e garantendo la scalabilità del sistema. Come descritto nel capitolo 4, troviamo quindi un backend scritto in Spring Boot per la logica di business e la gestione dei dati e un frontend in Angular per l'interfaccia utente. A questi è stato aggiunto un proxy intermedio il cui unico compito è gestire l'autenticazione e la comunicazione tra i due.

6.2.1 Architettura logica del sistema

Il nuovo componente, ovvero il proxy, è basato sempre su Spring Boot e si integra perfettamente nel sistema. È stato scelto Google come Identity Provider, data la sua elevata compatibilità. L'integrazione avviene tramite OAuth 2.0: gli utenti si autenticano al sistema utilizzando esclusivamente le proprie credenziali Google, senza che l'applicazione lato backend debba gestire password o dati sensibili all'interno del database. Questa scelta garantisce un'elevata sicurezza e un'esperienza utente già vista in altri scenari, che sia il più possibile user-friendly e non troppo difficile da usare.

6.2.2 Motivazioni della scelta del proxy personalizzato

Una delle scelte architettoniche più importanti è stata l'adozione di un proxy di autenticazione personalizzato, anziché utilizzare prodotti esistenti come OAuth2-Proxy. Questa scelta è stata fatta per soddisfare specifiche esigenze funzionali che le soluzioni convenzionali non potevano soddisfare. Il principale problema nell'utilizzo di un proxy OAuth2 è la gestione dei refresh token. Poter rinnovare automaticamente i token di accesso senza costringere l'utente a una nuova autenticazione è fondamentale nell'architettura moderna, che prevede sessioni di lunga durata e l'accesso da più dispositivi. OAuth2-Proxy propone funzionalità di session management e permette di utilizzare un session store basato sui cookie e il rinnovo degli access token tramite il refresh token.Tuttavia, nella pratica, si sono rivelati dei limiti molto importanti: in alcune release, quando il provider fornisce un nuovo refresh token, questo non viene aggiornato o il refresh non avviene in tempo utile e si deve di fatto ripetere il login.
Ci sono poi ulteriori limitazioni molto importanti per il progetto realizzato: per motivi di sicurezza, OAuth2-Proxy non invia il refresh token al backend. Sebbene

questo comportamento fosse giustificato dal principio di ridurre al minimo la superficie di attacco, nel contesto sperimentale era necessario utilizzare il refresh token per abilitare un meccanismo di autenticazione completamente privo di username e password, che evitasse di dover tornare ogni volta alla pagina di login di Google. L'impossibilità di accedere al refresh token ha reso difficile ottenere i vantaggi di un'autenticazione federata tradizionale in termini di trasparenza per l'utente e continuità della sessione. La gestione intelligente del ciclo di vita dei token, che include il salvataggio sicuro dei token di refresh, il loro rinnovo automatico prima della scadenza e la distribuzione trasparente dei nuovi token di accesso verso il frontend, è stata risolta dal proxy personalizzato creato. Questa soluzione riduce al minimo le interruzioni dell'esperienza utente e garantisce la continuità del servizio. Scegliere un proxy su misura ha permesso di ottenere numerosi benefici che vanno oltre la semplice risoluzione dei problemi legati ai refresh token. In primo luogo, viene garantito un controllo minuzioso sui processi di autenticazione con l'implementazione di logiche personalizzate per affrontare situazioni specifiche come il primo accesso e la gestione degli errori di autenticazione. Un altro vantaggio è senz'altro la possibilità di combinare l'autenticazione biometrica con WebAuthn nel tradizionale processo OAuth2, consentendo al proxy personalizzato di gestire entrambi i metodi di autenticazione. In questo modo, gli utenti possono accedere sia tramite il login con Google sia tramite la biometria, quindi con le impronte digitali o il riconoscimento facciale, a seconda del dispositivo utilizzato.

6.3 Flusso di autenticazione con WebAuthn e OAuth2

Tutto il flusso di autenticazione e la sua integrazione con WebAuthn sono gestiti da due controller principali: AuthController, che si occupa della registrazione e dell'integrazione con l'Identity Provider e con OAuth2.0, e WebAuthnController, che si occupa principalmente dell'integrazione con le API WebAuthn e della gestione delle credenziali, come la registrazione e l'autenticazione previste per le fasi di attestazione e asserzione. Come già accennato, WebAuthN è suddiviso in due fasi. Nella prima fase, l'utente registra un nuovo autenticatore presso il servizio web. Nella seconda fase, l'utente prova la proprietà di una chiave privata corrispondente a una chiave registrata. Nel proxy ho sviluppato un altro controller che si occupa invece della sua interazione con il backend: qui, infatti, troviamo le funzioni per firmare le richieste API verso il backend, inoltrare le richieste e estrarre il JWT token dai cookie o dall'header Authorization.

6.3.1 Registrazione utente con WebAuthn

Il flusso inizia con il processo di registrazione, necessario per associare l'impronta dell'utente a una credenziale e poter così accedere successivamente al sistema. La differenza con le altre modalità di accesso sicuro, come le passkey o altri SSO, sta proprio nella scelta di un approccio ibrido che unisce i benefici e la sicurezza garantiti dall'autenticazione **OAuth2** di **Google** con la tecnologia **WebAuthn**, in modo da garantire un accesso sicuro senza dover passare dalla pagina di accesso principale. In questo modo si è riusciti a sfruttare i vantaggi dell'*identity provider* esterno (**Google**) per la gestione dell'identità dell'utente. **WebAuthn**, invece, fornisce un livello di sicurezza aggiuntivo attraverso l'autenticazione biometrica locale.

Accesso con Google e redirezione

Tutto inizia quando un nuovo utente accede al sistema e clicca sul pulsante *"Accedi con Google"* presente nella pagina di login. Questo avvia una funzione di reindirizzamento verso l'endpoint `"/auth/google"` del proxy, che permette di rimandare l'utente alla pagina di accesso di **Google**.

```
1 loginWithGoogle() {
2     window.location.href = 'http://localhost:3000/auth/google';
3   }
```

Listing 6.1: loginWithGoogle

Callback e gestione dei filtri

Una volta completata la procedura di autenticazione presso l'*IdP*, l'**authorization server** esegue un'operazione di callback verso l'endpoint `/login/oauth2/code/google`, che viene automaticamente riconosciuto da **Spring Security** come endpoint **OAuth2**. Questo URL contiene i parametri `"code"` (codice di autorizzazione) e `"state"` (per la protezione CSRF). La richiesta viene intercettata da due filtri: **WebAuthnAuthenticationFilter**, un filtro personalizzato, e **OAuth2LoginAuthenticationFilter**, proprio di **Spring Security**, un framework modulare progettato per gestire la sicurezza nelle applicazioni Java. Il primo filtro, posizionato prima nella filter chain tramite `.addFilterBefore()`, verifica se sono attive delle sessioni di autenticazione biometrica. Non trovando sessioni attive in questa fase iniziale, il filtro passa la richiesta al filtro successivo senza intervenire. L'**OAuth2LoginAuthenticationFilter**, invece, riconosce il pattern dell'URL `/login/oauth2/code/google` e si attiva automaticamente per gestire lo scambio del token. In particolare, effettua due chiamate HTTP a **Google**: la prima verso l'endpoint `/token` per scambiare l'autorizzazione con

i token OAuth2 e la seconda verso l'endpoint `/userinfo` per recuperare i dati del profilo utente utilizzando l'*access token* appena ottenuto. Viene poi creato l'oggetto `authorizedClient`, che rappresenta un client OAuth2 autorizzato, e un `OAuth2AuthenticationToken` contenente l'`OAuth2User` personalizzato (configurato tramite `OAuth2UserService()`), che viene inserito nel **SecurityContext** di **Spring Security**, rendendo l'utente formalmente autenticato per l'applicazione.

Gestione dei token e salvataggio nel database

Dopo aver completato il filtro, la richiesta arriva alla funzione `loginSuccess` (Listing 6.2) nell'**AuthController**. In questa funzione, prima viene verificata l'autenticazione dell'utente da parte di **Google**. Vengono estratti i dati dell'utente, come l'indirizzo email, il nome e il cognome, e l'ID del soggetto di **Google**. Questi dati sono utili al sistema per essere salvati nel database. Quest'ultimo è una stringa alfanumerica generata da **Google** che rappresenta in modo univoco un utente autenticato tramite **OAuth 2.0**. Altri dati necessari sono il *refresh token* e l'*id token*. Il *refresh token* viene utilizzato per ottenere nuovi **JWT** da **Google** senza che l'utente debba effettuare nuovamente il login, mentre l'*id token* è una sorta di "documento d'identità digitale", un **JWT** firmato da **Google** che contiene claim standardizzati come `"sub"` (ID dell'utente), `"email"` e `"name"` e ha una scadenza tipica di un'ora. Prima di essere memorizzato nel cookie, viene sottoposto a crittografia ibrida **RSA+AES**. Questi dati vengono recepiti dall'`authorizedClient`, gestito dall'`OAuth2AuthorizedClientService`, che mantiene i token associati agli utenti autenticati. Tale servizio utilizza un'entità `OAuth2AuthenticationToken`, una classe di **Spring Security** che rappresenta un'autenticazione completata tramite **OAuth2**.

Come anticipato, questi dati vengono salvati nel database. In particolare, il *refresh token* viene criptato utilizzando algoritmi di cifratura simmetrica prima di essere memorizzato nel database, garantendo che, anche in caso di compromissione di quest'ultimo, il token rimanga inutilizzabile senza la chiave di decifratura. Viene poi indicata la prima autenticazione dell'utente, informazione utile per capire se l'utente non ha registrato l'autenticatore.

Primo login e registrazione WebAuthn

Successivamente, imposta l'*ID Token* di **Google** e reindirizza l'utente, in base al primo login, alla pagina /webauthn/register nel caso in cui l'utente debba registrare una credenziale, o alla pagina /`/webauthn/register` nel caso in cui debba registrare una credenziale, o alla pagina `/profile` se non è necessaria alcuna registrazione.

```
1  public void loginSuccess(HttpServletRequest request,
       HttpServletResponse response) throws Exception {
2          Authentication authentication = SecurityContextHolder.
       getContext().getAuthentication();
3
4          if (authentication == null || !authentication.
       isAuthenticated()) {
5              response.sendRedirect("http://localhost:4200/error");
6              return;
7          }
8
9          OAuth2User oauth2User = (OAuth2User) authentication.
       getPrincipal();
10         Map<String, Object> attributes = oauth2User.getAttributes
       ();
11
12         // Estrai i dati utente da Google
13         String email = (String) attributes.get("email");
14         String firstName = (String) attributes.get("given_name");
15         String lastName = (String) attributes.get("family_name");
16         String googleSub = (String) attributes.get("sub");
17
18         // Ottieni il refresh token e l'ID token
19         String refreshToken = null;
20         String idToken = null;
21
22         if (authentication instanceof OAuth2AuthenticationToken) {
23             OAuth2AuthenticationToken oauthToken = (
       OAuth2AuthenticationToken) authentication;
24             OAuth2AuthorizedClient authorizedClient =
       authorizedClientService.loadAuthorizedClient(
25                 oauthToken.getAuthorizedClientRegistrationId(),
26                 oauthToken.getName()
27             );
28
29             if (authorizedClient != null) {
30                 // Estrai refresh token
31                 if (authorizedClient.getRefreshToken() != null) {
32                     refreshToken = authorizedClient.
       getRefreshToken().getTokenValue();
33                 }
34
35                 // Estrai ID token (JWT di Google)
36                 if (authorizedClient.getAccessToken() != null) {
37                     // Per OpenID Connect, il principal dovrebbe
       essere OidcUser
38                     if (oauth2User instanceof OidcUser) {
39                         OidcUser oidcUser = (OidcUser) oauth2User;
```

```
40                          idToken = oidcUser.getIdToken().
    getTokenValue();
41                      }
42                  }
43              }
44          }
45
46          try {
47              // Salva l'utente nel database CON il refresh token
48              AppUser user = userService.findOrCreateUser(email,
    firstName, lastName, googleSub, refreshToken);
49
50              // Usa solo ID Token di Google
51              if (idToken != null) {
52                  setIdTokenCookie(response, idToken);
53              } else {
54                  System.err.println("ERRORE CRITICO: ID Token di
    Google non disponibile. Login fallito.");
55                  response.sendRedirect("http://localhost:4200/error
    ?message=google_token_unavailable");
56                  return;
57              }
58
59              // Reindirizza in base al primo login
60              if (user.isFirstLogin()) {
61                  response.sendRedirect("http://localhost:4200/
    webauthn/register");
62              } else {
63                  response.sendRedirect("http://localhost:4200/
    profile");
64              }
65
66          } catch (Exception e) {
67              System.err.println("Error during login success: " + e.
    getMessage());
68              response.sendRedirect("http://localhost:4200/error");
69          }
70          }
```

Listing 6.2: loginSuccess

Processo di attivazione dell'autenticazione biometrica

Poiché si tratta del primo accesso, l'utente viene reindirizzato alla pagina di registrazione **WebAuthn** (*"registerw"*) per configurare l'autenticazione biometrica. Il processo di registrazione è gestito dalla funzione `startRegistration`, che scambia informazioni con le funzioni del controller dedicato tramite gli endpoint appropriati. Il processo può essere suddiviso in cinque fasi distinte:

Inizialmente, la funzione effettua una richiesta **GET** all'endpoint `/api/webauthn/attestationOptions` (Listing 6.3) per ottenere le opzioni di registrazione dal server. A questo punto, il controller restituisce l'email dell'utente, genera una *challenge* crittografica univoca, la salva per la successiva verifica e crea le opzioni di registrazione **WebAuthn**, specificando l'entità *Relying Party* (che identifica il server), l'entità *User* (che contiene l'ID, l'email e il nome completo) e l'*authenticator selection*, composto da tre attributi: *attachment platform* (dispositivi integrati), *required user verification* (l'autenticatore deve verificare attivamente l'identità dell'utente prima di completare la registrazione) e *discouraged resident key* (si preferisce che la chiave privata non venga memorizzata in modo "resident" sull'autenticatore, ma che sia generata e usata al volo).

```
1  @GetMapping("/attestationOptions")
2  public ResponseEntity<CredentialCreationOptionsDTO> register(
     HttpServletRequest request) {
3          Authentication authentication = SecurityContextHolder.
     getContext().getAuthentication();
4
5          if (authentication != null && authentication.
     isAuthenticated() && authentication.getPrincipal() instanceof
     OAuth2User) {
6              OAuth2User oauth2User = (OAuth2User) authentication.
     getPrincipal();
7              Map<String, Object> attributes = oauth2User.
     getAttributes();
8
9              String email = (String) attributes.get("email");
10             AppUser user = userService.findUserEntityByEmail(email
     ).get();
11
12             Challenge challenge = challengeRepository.
     generateChallenge();
13             challengeRepository.saveChallenge(challenge, request);
14
15             PublicKeyCredentialRpEntity rpEntity = new
     PublicKeyCredentialRpEntity("localhost", "Proxy WebAuthn");
16
17             PublicKeyCredentialUserEntity userEntity = new
     PublicKeyCredentialUserEntity(
18                     user.getId().toString().getBytes(),
19                     user.getEmail(), // username
20                     user.getFullName() // display name
21             );
22
23             AuthenticatorSelectionCriteria authenticatorSelection
     = new AuthenticatorSelectionCriteria(
```

```
24                      AuthenticatorAttachment.PLATFORM,
25                      ResidentKeyRequirement.DISCOURAGED,
26                      UserVerificationRequirement.REQUIRED);
27
28              PublicKeyCredentialCreationOptions options = new
    PublicKeyCredentialCreationOptions(
29                      rpEntity,
30                      userEntity,
31                      challenge,
32                      publicKeyCredentialParameters,
33                      null, // timeout
34                      Collections.emptyList(),
35                      authenticatorSelection,
36                      AttestationConveyancePreference.NONE,
37                      null // extensions
38              );
39
40              CredentialCreationOptionsDTO dto = toDTO(options);
41              return ResponseEntity.ok(dto);
42
43          } else {
44              return ResponseEntity.status(HttpStatus.UNAUTHORIZED).
    body(null);
45          }
46      }
```

Listing 6.3: endpoint "/attestationOptions" nel WebAuthnController

Nella fase successiva, la funzione riceve le opzioni in formato **JSON** e le prepara per la **WebAuthn API**, decodificando i campi **Base64URL** (`challenge` e `user.id`) in **ArrayBuffer** con la funzione di utilità `base64urlToBuffer()`. Il browser chiama poi l'API **WebAuthn** per la registrazione tramite `navigator.credentials.create()` che attiva l'autenticazione del dispositivo. Dopo che l'utente appoggia il dito sul sensore, si passa alla fase successiva, in cui i dati vengono preparati per l'invio al server, che si occuperà di storicizzarli per le prossime verifiche, convertendo i campi binari (`rawId`, `clientDataJSON`, `attestationObject`) in **Base64URL** tramite `bufferToBase64url()`.
L'oggetto credenziale viene infine inviato all'endpoint **POST**/api/webauthn/register (Listing 6.4). In questo caso, il proxy recupera la *challenge* nella sessione, utilizza il `WebAuthnManager` (fornito dalla libreria **WebAuthn4J**) per validare l'attestazione, configura i parametri specificando i dati necessari, quali l'*origin*, il *server propriety* e i requisiti di verifica, e infine estrae il `credentialId` e lo codifica. Insieme a quest'ultimo, la credenziale viene salvata nel database, che risulterà utile per recuperarla successivamente al login.

```
1   @PostMapping("/register")
```

```
2      public ResponseEntity<String> verifyRegistration(@RequestBody
       Map<String, Object> requestBody,
3              HttpServletRequest request) throws IOException {
4
5          Authentication authentication = SecurityContextHolder.
       getContext().getAuthentication();
6
7          if (authentication != null && authentication.
       isAuthenticated() && authentication.getPrincipal() instanceof
       OAuth2User) {
8              OAuth2User oauth2User = (OAuth2User) authentication.
       getPrincipal();
9              Map<String, Object> attributes = oauth2User.
       getAttributes();
10
11             String email = (String) attributes.get("email");
12             String json = new ObjectMapper().writeValueAsString(
       requestBody.get("registrationResponseJSON"));
13
14             Challenge challenge = challengeRepository.
       loadChallenge(request);
15             if (challenge == null) {
16                 return ResponseEntity.status(HttpStatus.
       BAD_REQUEST).body("Challenge non trovato");
17             }
18
19             Origin origin = new Origin("http://localhost:4200");
20             ServerProperty serverProperty = new ServerProperty(
       origin, "localhost", challenge);
21
22             boolean userVerificationRequired = false;
23             boolean userPresenceRequired = true;
24
25             RegistrationParameters registrationParameters = new
       RegistrationParameters(serverProperty,
26                     publicKeyCredentialParameters,
27                     userVerificationRequired, userPresenceRequired
       );
28
29             RegistrationData registrationData;
30
31             try {
32                 registrationData = webAuthnManager.
       parseRegistrationResponseJSON(json);
33                 webAuthnManager.verify(registrationData,
       registrationParameters);
34             } catch (DataConversionException |
       VerificationException e) {
35                 e.printStackTrace();
```

```
36                  return ResponseEntity.status(HttpStatus.
    BAD_REQUEST).body("Verifica fallita: " + e.getMessage());
37              }
38
39              AppUser user = userService.findUserEntityByEmail(email
    )
40                      .orElseThrow(() -> new RuntimeException("
    Utente non trovato"));
41
42              // Estrai il credential ID dai dati di registrazione
43              String credentialId = com.webauthn4j.util.
    Base64UrlUtil.encodeToString(
44                  registrationData.getAttestationObject().
    getAuthenticatorData().getAttestedCredentialData().
    getCredentialId()
45              );
46
47              // Salva i dati di registrazione usando il
    RegistrationService
48              registrationService.saveRegistrationData(user,
    registrationData, credentialId);
49
50              return ResponseEntity.ok("Registrazione completata con
     successo");
51          } else {
52              return ResponseEntity.status(HttpStatus.UNAUTHORIZED).
    body("Utente non autenticato");
53          }
54      }
```

Listing 6.4: endpoint "/register" nel WebAuthnController

6.3.2 Login biometrico e validazione Google

Al termine della procedura di registrazione, l'accesso al sistema è possibile unicamente tramite impronta digitale. Nella pagina di login è presente un pulsante che avvia la procedura. In particolare, il pulsante è associato alla funzione `loginWithTouchID` (Listing 6.5), scelta per semplicità e per corrispondere al dispositivo utilizzato, ma potrebbe avere un nome diverso. Anche in questa funzione avviene una comunicazione con il controller menzionato in precedenza e il processo viene suddiviso in cinque fasi distinte.

```
1  async loginWithTouchID(): Promise<void> {
2      this.isLoadingTouchID = true;
3
4      try {
5
```

```
6       // Step 1: Ottieni le opzioni di autenticazione dal server
7       const options = await fetch("http://localhost:3000/api/
    webauthn/authenticationOptions", {
8         credentials: 'include'
9       });
10
11      if (!options.ok) {
12        throw new Error(`Errore nel recupero delle opzioni: ${
    options.status}`);
13      }
14
15      const data = await options.json();
16
17      // Step 2: Chiama la WebAuthn API per l'autenticazione
18      const assertion = await navigator.credentials.get({
19        publicKey: {
20          challenge: base64urlToBuffer(data.challenge.value),
21          rpId: data.rpId,
22          allowCredentials: data.allowCredentials,
23          userVerification: data.userVerification,
24          timeout: data.timeout
25        }
26      }) as PublicKeyCredential;
27
28      if (!assertion) {
29        throw new Error('Autenticazione TouchID annullata o
    fallita');
30      }
31
32
33      // Step 3: Converte l'assertion nel formato corretto per
    WebAuthn4J
34      const authenticationResponse = {
35        id: assertion.id,
36        type: assertion.type,
37        rawId: bufferToBase64url(assertion.rawId),
38        response: {
39          clientDataJSON: bufferToBase64url((assertion.response as
     AuthenticatorAssertionResponse).clientDataJSON),
40          authenticatorData: bufferToBase64url((assertion.response
     as AuthenticatorAssertionResponse).authenticatorData),
41          signature: bufferToBase64url((assertion.response as
    AuthenticatorAssertionResponse).signature),
42          userHandle: (assertion.response as
    AuthenticatorAssertionResponse).userHandle ?
43                      bufferToBase64url((assertion.response as
    AuthenticatorAssertionResponse).userHandle!) : null
44        }
45      };
```

```
46
47
48        // Step 4: Invia i dati al server per la verifica
49        const response = await fetch('http://localhost:3000/api/
      webauthn/login', {
50          method: 'POST',
51          body: JSON.stringify({ authenticationResponse }),
52          headers: { 'Content-Type': 'application/json' },
53          credentials: 'include'
54        });
55
56        if (!response.ok) {
57          const errorText = await response.text();
58          throw new Error(`Login TouchID fallito: ${errorText}`);
59        }
60
61        console.log("Login WebAuthn completato con successo!");
62
63        // Step 5: Aggiorna lo stato di autenticazione e reindirizza
64        this.authService.updateAuthenticationStatus(true);
65        this.router.navigate(['/profile']);
66
67      } catch (error: any) {
68        console.error('Errore durante il login TouchID:', error);
69
70        // Gestione errori
71        let errorMessage = 'Errore sconosciuto durante il login
      TouchID.';
72
73        if (error.name === 'NotAllowedError') {
74          errorMessage = 'Login TouchID annullato dall\'utente.';
75        } else if (error.name === 'NotSupportedError') {
76          errorMessage = 'TouchID non è supportato su questo
      dispositivo.';
77        } else if (error.name === 'SecurityError') {
78          errorMessage = 'Errore di sicurezza. Connessione HTTPS
      richiesta.';
79        } else if (error.name === 'InvalidStateError') {
80          errorMessage = 'Nessuna credenziale TouchID registrata per
       questo account.';
81        } else if (error.message) {
82          errorMessage = error.message;
83        }
84
85        alert(errorMessage);
86
87      } finally {
88        this.isLoadingTouchID = false;
89      }
```

```
90    }
```

Listing 6.5: loginWithTouchID

Richiesta delle opzioni di autenticazione

Come nella fase di registrazione, nella prima fase vengono ottenute le opzioni dal server, ma in questo caso si tratta di *opzioni di autenticazione*. Con una richiesta **GET**, il controller genera una nuova *sfida crittografica* e la salva nella sessione HTTP (Listing 6.6), ottenuta dal frontend. Questa risposta include la *challenge*, l'**RP ID** (identificatore del *Relying Party*, ovvero in questo caso *localhost*), l'attributo `userVerification` impostato su **REQUIRED** per garantire la verifica biometrica e una lista di `allowCredentials` che specifica quali credenziali sono accettate per quell'utente. Ogni elemento rappresenta un **ID credenziale** precedentemente registrato.

```
1  @GetMapping("/authenticationOptions")
2      public ResponseEntity<PublicKeyCredentialRequestOptions>
       getAssertionOptions(HttpServletRequest request) {
3          Challenge challenge = challengeRepository.
       generateChallenge();
4          challengeRepository.saveChallenge(challenge, request);
5
6          PublicKeyCredentialRequestOptions options = new
       PublicKeyCredentialRequestOptions(
7                  challenge,
8                  null, // timeout
9                  "localhost", // rpId
10                 Collections.emptyList(), // allowCredentials
11                 UserVerificationRequirement.REQUIRED,
12                 null // extensions
13         );
14
15         return ResponseEntity.ok(options);
16     }
```

Listing 6.6: endpoint "/authenticationOptions" nel WebAuthnController

Attivazione dell'autenticatore

Anche qui, il frontend converte i parametri in un **ArrayBuffer**, preparandoli per la API `navigator.credentials.get()`. La chiamata attiva l'autenticatore del dispositivo che verifica l'identità biometrica, genera una firma crittografica utilizzando la chiave privata associata alla credenziale e restituisce un oggetto **PublicKeyCredential** contenente i dati di autenticazione.

Conversione e preparazione dei dati

Nella fase successiva, i dati dell'asserzione vengono convertiti in formato **JSON** compatibile con la libreria **WebAuthn4J**, codificandoli in **Base64URL**. I dati convertiti sono: `clientDataJson` (metadati del client inclusi origin e challenge), `authenticatorData` (informazioni dell'autenticatore e del *signature count*, un contatore che tiene traccia delle volte in cui è stata effettuata un'autenticazione con quella credenziale, utile per la protezione contro gli attacchi di tipo replay), la `signature` (firma crittografica generata dall'autenticatore) e lo `userHandle` (identificatore opzionale dell'utente).

Verifica lato server

L'endpoint `/api/webauthn/login` (Listing 6.7)riceve i dati e procede con la verifica. La funzione nel proxy, infatti, si occupa prima di deserializzare i dati grazie al `WebAuthnManager` reso disponibile dalla libreria **WebAuthn4J**. In questo modo, il sistema cerca la `credentialRecord` tramite il `credentialId` nel database, la associa all'utente corrispondente, recupera e valida la *challenge* salvata nella sessione e il `WebAuthnManager` convalida la firma utilizzando la chiave pubblica memorizzata, incrementando poi il *signature count* associato.

```
1  @PostMapping("/login")
2      public ResponseEntity<String> verifyLogin(@RequestBody Map<
       String, Object> requestBody,
3              HttpServletRequest request, HttpSession session,
       HttpServletResponse response) throws JsonProcessingException {
4
5          String authenticationResponseJSON = new ObjectMapper()
6                  .writeValueAsString(requestBody.get("
       authenticationResponse"));
7
8          AuthenticationData authenticationData;
9          try {
10             authenticationData = webAuthnManager.
       parseAuthenticationResponseJSON(authenticationResponseJSON);
11         } catch (DataConversionException e) {
12             System.err.println("Errore nel parsing del JSON
       WebAuthn: " + e.getMessage());
13             e.printStackTrace();
14             return ResponseEntity.status(HttpStatus.BAD_REQUEST).
       body("Parsing fallito: " + e.getMessage());
15         } catch (Exception e) {
16             System.err.println("Errore generico nel parsing: " + e
       .getMessage());
17             e.printStackTrace();
```

```
18              return ResponseEntity.status(HttpStatus.BAD_REQUEST).
    body("Errore imprevisto: " + e.getMessage());
19          }
20
21          byte[] credentialId = authenticationData.getCredentialId()
    ;
22          String credentialIdBase64 = com.webauthn4j.util.
    Base64UrlUtil.encodeToString(credentialId);
23
24          // Recupera la CredentialRecord usando il
    RegistrationService
25          Optional<CredentialRecord> webAuthnCredential =
    registrationService.getCredentialRecordByCredentialId(
    credentialIdBase64);
26
27          if (webAuthnCredential.isEmpty()) {
28              return ResponseEntity.status(HttpStatus.NOT_FOUND).
    body("Credential non trovata per ID: " + credentialIdBase64);
29          }
30
31          // Trova l'utente associato alla credenziale
32          Optional<AppUser> user = registrationService.
    findUserByCredentialId(credentialIdBase64);
33
34          if (user.isEmpty()) {
35              return ResponseEntity.status(HttpStatus.NOT_FOUND).
    body("Utente non trovato per ID: " + credentialIdBase64);
36          }
37
38          Challenge loadedChallenge = challengeRepository.
    loadChallenge(request);
39
40          ServerProperty serverProperty = new ServerProperty(
41                  new Origin("http://localhost:4200"), "localhost",
    loadedChallenge);
42
43          AuthenticationParameters authenticationParameters = new
    AuthenticationParameters(
44
            serverProperty,
45
            webAuthnCredential.get(),
46
            List.of(credentialId), // allowCredentials
47
            true, //userVerificationRequired
48
            true // userPresenceRequired
49          );
```

```
50          try {
51              webAuthnManager.verify(authenticationData,
    authenticationParameters);
52          } catch (VerificationException e) {
53              e.printStackTrace();
54              return ResponseEntity.status(HttpStatus.BAD_REQUEST).
    body("Verifica fallita: " + e.getMessage());
55          }
56
57          // Aggiorna il signature count
58          try {
59              long newSignatureCount = authenticationData.
    getAuthenticatorData().getSignCount();
60              registrationService.updateSignatureCount(
    credentialIdBase64, newSignatureCount);
61          } catch (Exception e) {
62              System.err.println("Errore nell'aggiornamento del
    signature count: " + e.getMessage());
63          }
64
65          // Salva il fatto che l'utente è autenticato nella
    sessione
66          session.setAttribute("loggedIn", true);
67          session.setAttribute("userId", user.get().getId());
68
69          // Crea un'autenticazione Spring Security per WebAuthn
70          createWebAuthnAuthentication(user.get());
71
72          // Tenta di ottenere un ID Token usando il refresh token
    salvato (decrittografato)
73          String refreshToken = userService.getDecryptedRefreshToken
    (user.get());
74          if (refreshToken != null && !refreshToken.trim().isEmpty()
    ) {
75              try {
76                  String idToken = googleTokenService.refreshIdToken
    (refreshToken);
77                  if (idToken != null) {
78                      setIdTokenCookie(response, idToken);
79                  } else {
80                      System.err.println("ERRORE CRITICO:
    Impossibile ottenere ID Token, refresh token potrebbe essere
    scaduto");
81                      return ResponseEntity.status(HttpStatus.
    INTERNAL_SERVER_ERROR)
82                              .body("Errore nell'autenticazione: token
    Google non disponibile");
83                  }
84              } catch (Exception e) {
```

```
85                  System.err.println("Errore nel refresh dell'ID
    Token: " + e.getMessage());
86                  return ResponseEntity.status(HttpStatus.
    INTERNAL_SERVER_ERROR)
87                          .body("Errore nell'autenticazione: " + e.
    getMessage());
88              }
89          } else {
90              System.err.println("ERRORE CRITICO: Nessun refresh
    token disponibile o decrittografia fallita per questo utente");
91              return ResponseEntity.status(HttpStatus.
    INTERNAL_SERVER_ERROR)
92                      .body("Errore nell'autenticazione: refresh token
    Google non disponibile");
93          }
94
95          return ResponseEntity.ok("Login effettuato con successo");
96      }
```

Listing 6.7: endpoint "/login" nel WebAuthnController

Compatibilità con OAuth2 e generazione dei token

A questo punto, l'utente è loggato nel sistema e gli attributi vengono marcati nella sessione. Vengono inoltre generati dei *token fittizi* per garantire la compatibilità con l'architettura **OAuth2** esistente. La creazione di un `OAuth2AuthenticationToken` fittizio, infatti, garantisce la piena compatibilità con i filtri di sicurezza di **Spring Security**, permettendo al sistema di trattare l'autenticazione **WebAuthn** come se fosse un normale login **OAuth2**.

Gestione dei token di Google

In seguito, il sistema decrittografa il *refresh token* di **Google** salvato durante la registrazione iniziale e richiede un nuovo *ID token* tramite il servizio `googleTokenService` creato. La logica appena descritta è implementata in una funzione dedicata, che si occupa di costruire la richiesta HTTP e chiamare il Google Token Endpoint ufficiale per ottenere l'ID token.

Misure di sicurezza implementate

È importante notare che durante tutto il processo, vengono implementate diverse misure di sicurezza: la validazione dell'*origin* (`http://localhost:4200`) per prevenire attacchi CSRF, la verifica temporale della *challenge* per evitare attacchi replay, e la validazione del *signature count* che deve essere incrementale rispetto all'ultimo valore memorizzato.

6.3.3 Gestione dei Token e Sicurezza

Per un'ulteriore sicurezza, il proxy, dopo aver ottenuto il cookie da Google (sia tramite login OAuth2 diretto che tramite refresh token dopo il login biometrico), applica un livello di crittografia ibrida per proteggere il token durante la trasmissione tra il proxy e il backend e durante lo storage nei cookie (Come mostrato in 6.8). Infatti, il **JWT** viene sottoposto a tre tipi di crittografia:

1. **Crittografia AES-256**: il **JWT** viene criptato con una chiave simmetrica generata dinamicamente;
2. **Crittografia RSA-2048**: la chiave AES viene a sua volta criptata con la chiave privata RSA del proxy;
3. **Firma digitale**: il **JWT** originale viene firmato digitalmente per garantire l'integrità.

In questo modo, viene costruito un token conforme allo standard dell' *encrypted JWT*.

Di conseguenza, il **JWT** criptato viene archiviato in un cookie con attributi di sicurezza avanzati:

- **HttpOnly** per renderlo inaccessibile al JavaScript e prevenire attacchi XSS,
- **Secure** per trasmissione solo su HTTPS,
- **SameSite=None** al fine di evitare anche attacchi CSRF.

Così, ogni volta che viene fatta una richiesta all'API, il proxy legge prima il **JWT** dal cookie e poi lo firma di nuovo con la sua chiave privata, garantendogli l'autenticità e l'integrità della richiesta.

```
1  /**
2       * Cripta il JWT per il frontend usando crittografia ibrida (
       RSA + AES)
3       * Garantisce autenticità, integrità e riservatezza
4       */
5      public String encryptJwtForFrontend(String jwt) {
6          try {
7              // 1. Cripta il JWT con AES (efficiente per dati
       grandi)
8              Cipher aesCipher = Cipher.getInstance(
       AES_TRANSFORMATION);
9              aesCipher.init(Cipher.ENCRYPT_MODE, aesKey);
10             byte[] encryptedJwt = aesCipher.doFinal(jwt.getBytes("
       UTF-8"));
```

```
11
12              // 2. Cripta la chiave AES con RSA (sicuro per chiavi)
13              Cipher rsaCipher = Cipher.getInstance(
    RSA_TRANSFORMATION);
14              rsaCipher.init(Cipher.ENCRYPT_MODE, rsaKeyPair.
    getPrivate());
15              byte[] encryptedAesKey = rsaCipher.doFinal(aesKey.
    getEncoded());
16
17              // 3. Crea signature del JWT originale per garantire
    autenticità
18              Signature signature = Signature.getInstance("
    SHA256withRSA");
19              signature.initSign(rsaKeyPair.getPrivate());
20              signature.update(jwt.getBytes("UTF-8"));
21              byte[] digitalSignature = signature.sign();
22
23              // 4. Combina tutto in un formato serializzato
24              String encryptedData = Base64.getEncoder().
    encodeToString(encryptedJwt);
25              String encryptedKey = Base64.getEncoder().
    encodeToString(encryptedAesKey);
26              String signatureB64 = Base64.getEncoder().
    encodeToString(digitalSignature);
27
28              // Formato: [encrypted_data].[encrypted_key].[
    signature]
29              return encryptedData + "." + encryptedKey + "." +
    signatureB64;
30
31          } catch (Exception e) {
32              System.err.println("Errore nella crittografia del JWT:
     " + e.getMessage());
33              throw new RuntimeException("Errore nella crittografia
    del JWT per frontend", e);
34          }
35      }
```

Listing 6.8: Funzione di crittografia del token prima di settare il cookie

6.3.4 Accesso alle API protette del backend

Come già detto, il frontend è stato progettato per visualizzare gli appuntamenti di una persona.
Quindi, se il frontend effettua una chiamata API per ricevere risorse protette nel backend, come ad esempio `GET` `/api/appointments` per accedere alla lista degli appuntamenti, il proxy esegue un processo di validazione prima di inoltrare la

richiesta al frontend, che prevede una serie di passaggi (Listing 6.9).

```
/**
     * FUNZIONE PRINCIPALE: Proxy Sicuro per Richieste Backend
     *
     * Implementa il flusso di sicurezza completo per l'autenticazione JWT:
     * 1. Estrazione automatica del JWT criptato
     * 2. Decrittografia con chiavi RSA+AES
     * 3. Validazione e verifica dell'integrità
     * 4. Firma digitale per il backend
     * 5. Trasmissione sicura
     *
     * @param request Richiesta HTTP dal frontend con JWT criptato
     * @param endpoint Endpoint del backend da chiamare
     * @param method Metodo HTTP (GET, POST, PUT, DELETE)
     * @param requestBody Payload della richiesta (per POST/PUT)
     * @return Risposta del backend o errore di autenticazione
     */
    private ResponseEntity<?> secureProxyRequest(HttpServletRequest request, String endpoint,
                                                 HttpMethod method, Object requestBody) {
        try {
            // FASE 1: ESTRAZIONE DEL JWT CRIPTATO
            // Estrae automaticamente il JWT dal cookie HTTP-only 'id_token'
            // Il cookie contiene il JWT criptato inviato dal frontend Angular
            String encryptedJwt = extractJwtFromSecureCookie(request);
            if (encryptedJwt == null) {
                return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                        .body(createErrorResponse("JWT_NOT_FOUND",
                                "Token di autenticazione non trovato"));
            }

            // FASE 2: DECRITTOGRAFIA DEL JWT
            // Utilizza il servizio di crittografia per decrittare il JWT
            // Implementa decrittografia ibrida RSA+AES per massima sicurezza
            String originalJwt;
            if (isEncryptedJwt(encryptedJwt)) {
                try {
                    // Decrittografia con chiavi RSA+AES del proxy
```

```
37                      originalJwt = jwtCryptographyService.
    decryptJwtFromFrontend(encryptedJwt);
38                      if (originalJwt == null) {
39                          return ResponseEntity.status(HttpStatus.
    UNAUTHORIZED)
40                                  .body(createErrorResponse("
    DECRYPTION_FAILED","Impossibile decrittare il token"));
41                      }
42                  } catch (Exception e) {
43                      // Gestione errori di decrittografia (chiavi
    cambiate, token corrotto)
44                      return ResponseEntity.status(HttpStatus.
    UNAUTHORIZED)
45                              .header("X-Auth-Error", "
    JWT_DECRYPTION_FAILED")
46                              .body(createErrorResponse("
    DECRYPTION_ERROR",
47                                      "Token corrotto o chiavi non
    valide. Rieffettuare il login."));
48                  }
49              } else {
50                  // JWT non criptato (modalità di fallback per
    compatibilità)
51                  originalJwt = encryptedJwt;
52              }
53
54              // FASE 3: VERIFICA DI VALIDITÀ
55              // Il JWT originale viene utilizzato per verificare:
56              // - Scadenza del token
57              // - Integrità della struttura
58              // - Validità della firma Google (se applicabile)
59              if (!isValidJwtStructure(originalJwt)) {
60                  return ResponseEntity.status(HttpStatus.
    UNAUTHORIZED)
61                          .body(createErrorResponse("INVALID_JWT",
62                                  "Struttura JWT non valida"));
63              }
64
65              // FASE 4: FIRMA PER IL BACKEND
66              // Genera una nuova firma digitale specifica per il
    backend
67              // Questo garantisce che solo il proxy autorizzato
    possa comunicare col backend
68              String backendSignedJwt = jwtCryptographyService.
    signJwtForBackend(originalJwt);
69
70              // FASE 5: TRASMISSIONE SICURA AL BACKEND
71              // Prepara la richiesta HTTP con il JWT firmato
```

```
72              HttpHeaders headers = prepareSecureHeaders(
    backendSignedJwt);
73              copyClientHeaders(request, headers);
74
75              // Crea l'entità HTTP appropriata in base al metodo
76              HttpEntity<?> entity = createHttpEntity(method,
    requestBody, headers);
77
78              // Esegue la chiamata sicura al backend
79              return executeSecureBackendCall(endpoint, method,
    entity);
80
81          } catch (Exception e) {
82              // Gestione centralizzata degli errori del proxy
83              return ResponseEntity.status(HttpStatus.
    INTERNAL_SERVER_ERROR)
84                      .body(createErrorResponse("PROXY_ERROR",
85                              "Errore interno del proxy: " + e.
    getMessage()));
86          }
87      }
```

Listing 6.9: Flusso di autenticazione JWT del proxy

In primo luogo, il proxy estrae il **JWT** criptato dal cookie *HttpOnly* inviato dal frontend, dopodiché controlla che il **JWT** ricevuto sia effettivamente criptato e, in tal caso, lo decrittografa implementando una decrittografia ibrida, come per la crittografia implementata in precedenza, al fine di garantire la massima sicurezza.

Superato questo passaggio, il **JWT** originale viene utilizzato per verificare la scadenza del token e l'integrità della struttura. Fatto questo, il proxy genera una nuova firma digitale specifica per il backend, in modo che solo il proxy autorizzato possa comunicare con esso. Prepara quindi la richiesta **HTTP** con il **JWT** firmato e la inoltra al backend.

Prima di restituire le risorse richieste, però, il backend applica una doppia validazione (Listing 6.10): prima verifica se il **JWT** è firmato dal proxy con la sua chiave pubblica (per confermare la provenienza) e poi valida il **JWT** originale con la chiave pubblica di Google (per la validità dell'utente), ottenendo le informazioni necessarie, come l'e-mail, per poter recuperare gli appuntamenti associati.

```
1 // Validazione della firma del proxy
2 String originalJWT = jwtValidationService.verifyProxySignedJwt(
    signedJWT);
3
4 // Validazione del JWT Google originale
```

```
5 String userEmail = jwtValidationService.validateGoogleIdToken(
    originalJWT);
```

Listing 6.10: Doppia validazione del backend

Se le verifiche vanno a buon fine, il backend restituisce le risorse necessarie al frontend in una lista e delle entità sviluppate appositamente per questo caso.

6.3.5 Diagramma di sequenza

Il flusso completo di autenticazione, dalla registrazione iniziale al login biometrico fino all'accesso alle API, è illustrato nel diagramma mostrato in figura 6.1.
Questo flusso garantisce un'esperienza utente fluida, riuscendo a mantenere elevati standard e metodi di sicurezza attraverso l'integrazione di autenticazione biometrica, crittografia avanzata e validazione multi-livello dei token, e raggiungendo con successo l'obiettivo prefissato.

6.4 Esperienza utente con autenticazione biometrica

L'implementazione della biometria all'interno di questo caso d'uso ha una duplice finalità: da una parte punta a garantire ed incrementare il livello di sicurezza delle applicazioni, fornendo un ulteriore elemento su cui basare il riconoscimento dell'utente e consentirgli un accesso esclusivo alle proprie risorse. La novità sta proprio nel fatto che i token rilasciati sono garantiti da un IdP, senza che la Power App ne gestisca il rilascio, in altre parole la nostra identità non è soltanto quella che dichiariamo alla Power App, ma anche quella che dichiariamo ad un terzo attendibile e importante come lo è Google. Dall'altra si voleva creare un'esperienza utente semplice, per spingerne l'utilizzo e sfruttare tutti i benefici di una Progressive Web App rispetto ad un'app nativa, evitando che gli utenti debbano ricordare password o metodi troppo articolati e che sia sufficiente che essi utilizzino semplicemente il proprio dito o il proprio volto. Per ottenere ciò, l'applicazione deve avere un'interfaccia semplice e intuitiva, ma soprattutto rapida e non ambigua, che accompagni l'utente durante tutto il flusso senza fargli percepire che sta utilizzando un'app web piuttosto che un'app nativa.
Lo scopo di questa sezione è pertanto quello di mostrare visualmente il flusso, senza addentrarsi nuovamente negli aspetti tecnologici già analizzati, ma anche quello di valutare l'impatto dell'autenticazione biometrica in termini di usabilità, velocità e semplicità.

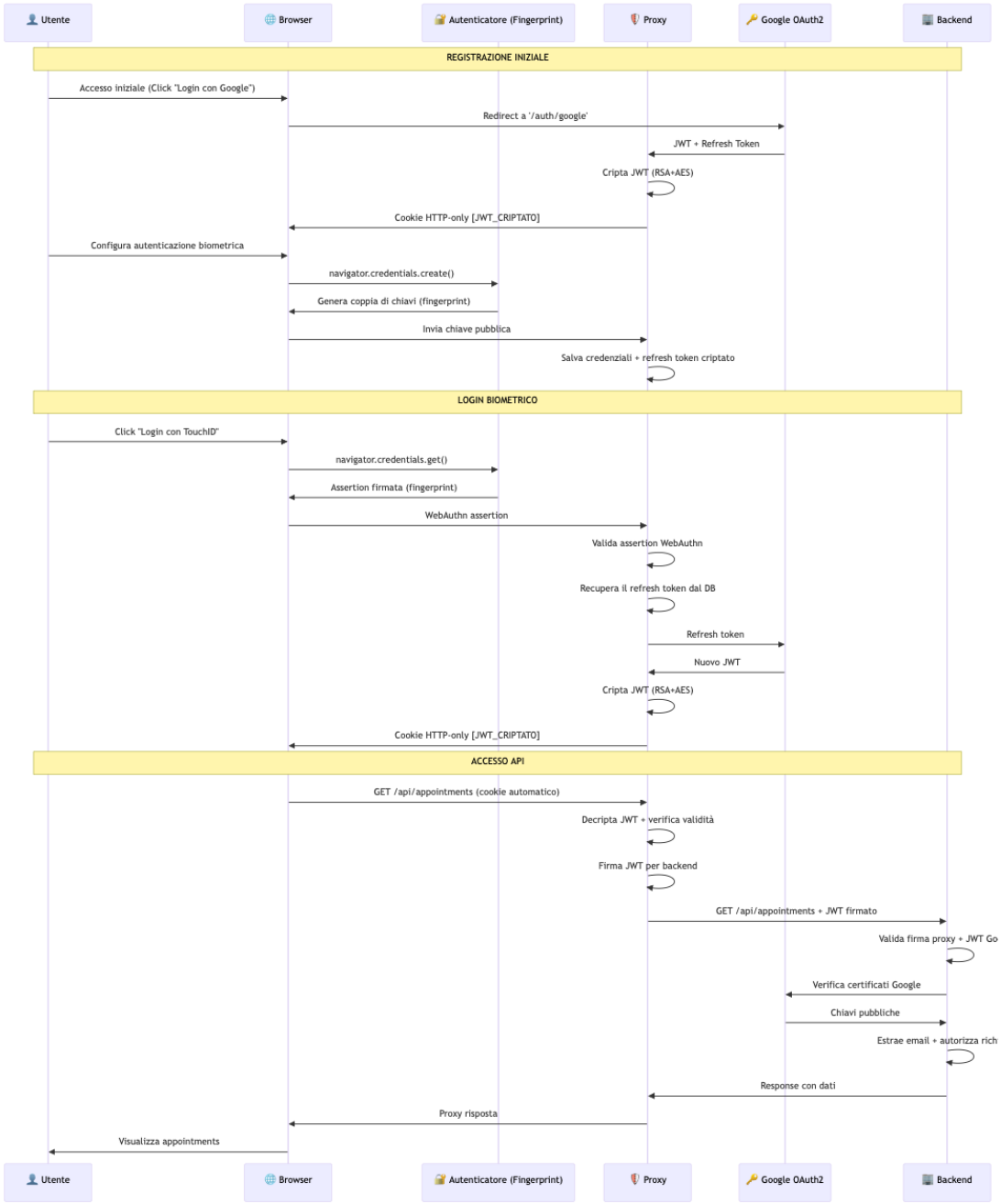

Figura 6.1: Flusso di autenticazione completo

6.4.1 Il Flusso Completo

È possibile avviare il processo di autenticazione tramite l'Identity Provider (IdP) dalla schermata di login (Figura 5.3) premendo sul bottone "Login con Google",

che indirizzerà l'utente alla pagina di accesso di Google. Da qui si potrà scegliere un utente che si è già utilizzato in precedenza oppure accedere con un nuovo utente inserendo le proprie credenziali. Al termine dell'autenticazione con Google il sistema verifica se per quell'utente abbia già registrato una credenziale biometrica. Nel caso non sia disponibile verrà mostrata la schermata di registrazione dell'impronta digitale, (Fig. 6.2) realizzata tramite le API WebAuthn. Premendo "Registra impronta biometrica" l'applicazione chiamerà l'API che aprirà la modale di sistema (Fig. 6.3) e darà inizio alla procedura di rilevamento del dito sul sensore. Appoggiato il dito a completamento della fase di associazione dell'impronta, la PWA aggiornerà la schermata confermando l'esito positivo (Fig. 6.4) e da quel momento l'utente potrà accedere al sistema utilizzando solo ed esclusivamente la propria impronta digitale.
Basterà infatti premere il tasto "Login con Touch ID" per riavviare la procedura che farà mostrare nuovamente la modale di riconoscimento (Fig. 6.5) e, in caso di esito positivo, l'utente verrà reindirizzato automaticamente alla propria dashboard (Fig. 5.4) .
Il flusso così concluso risulta fluido ed intuitivo ma anche semplice perchè permette di completare la procedura anche ad utenti poco avvezzi con le tecnologie digitali.

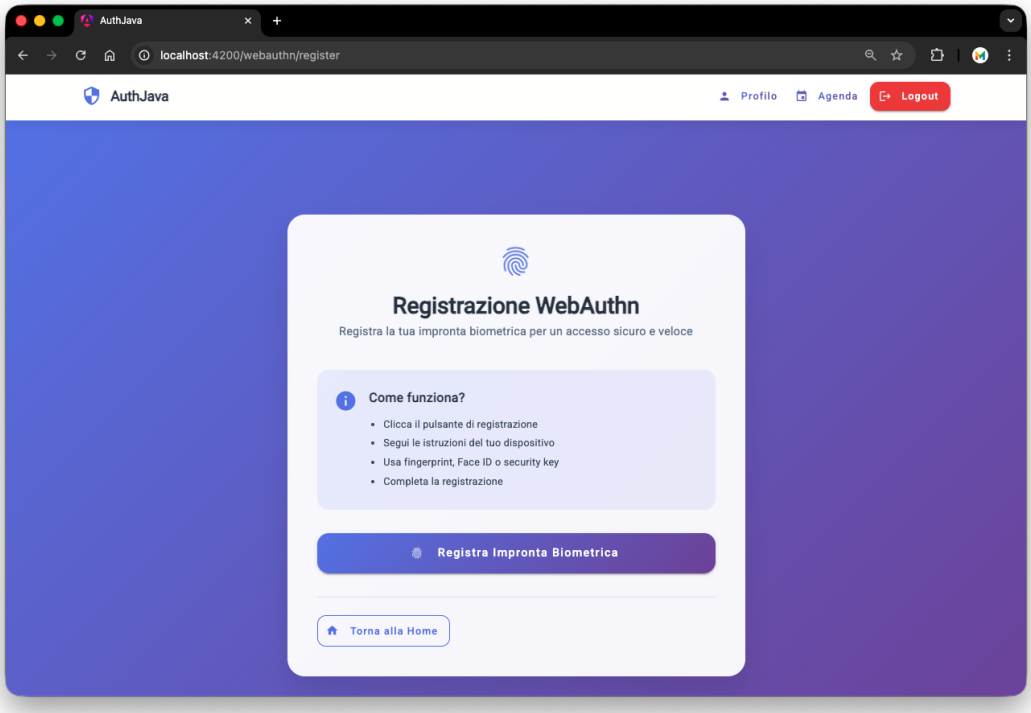

Figura 6.2: Pagina di registrazione dell'impronta per associarla al proprio account

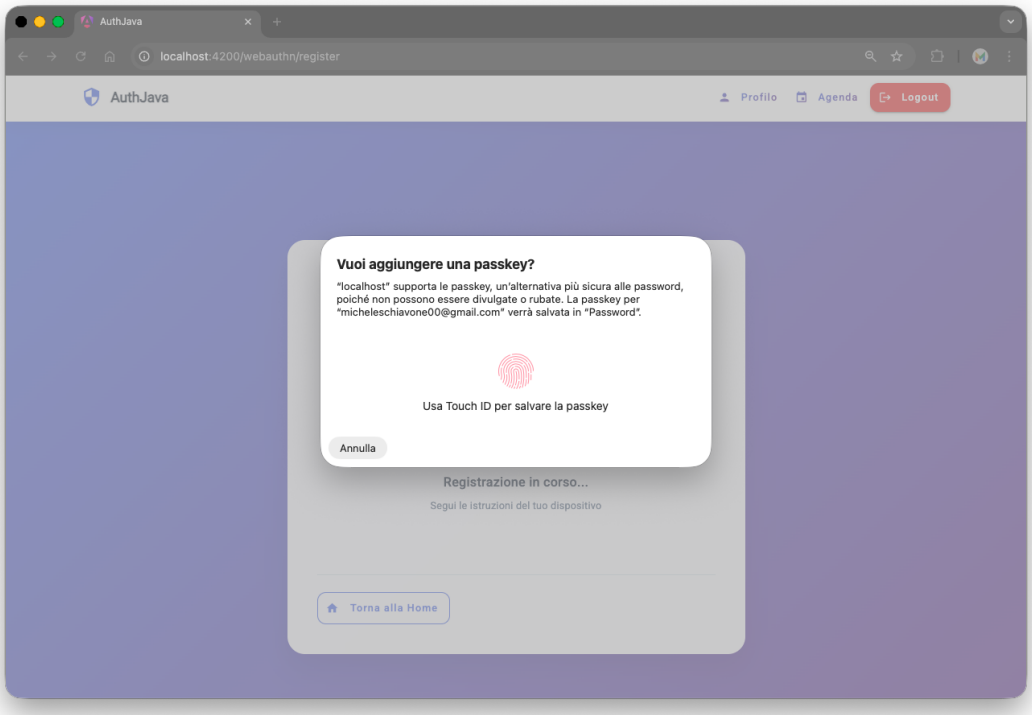

Figura 6.3: Modale di sistema in attesa di un'impronta

6.4.2 Valutazione dell'esperienza utente e confronto con login tradizionale

L'esperienza utente è fluida e molto reattiva. Dalle analisi eseguite, il tempo necessario per accedere utilizzando un metodo di autenticazione come l'impronta digitale è di soli 5 secondi, durante i quali è sufficiente cliccare il pulsante di accesso con Google e successivamente appoggiare il dito sul sensore. Se invece si accede con Google inserendo username e password, il tempo varia da account a account in base alla lunghezza dell'email o della password e alla capacità dell'utente di digitare rapidamente. In media, si avvicina ai 20 secondi, ma sui computer personali Google evita di far inserire nuovamente password e email, quindi il tempo impiegato è soltanto quello di cliccare il pulsante di accesso e inserire l'account con cui entrare, che si avvicina approssimativamente agli 8 secondi, risultando comunque un miglioramento. Infatti, il tempo di risposta delle chiamate di rete coinvolte è di soli 324 ms.

A questi, però, bisogna aggiungere altri miglioramenti che non riguardano il tempo di risposta e di reattività, come la riduzione del carico cognitivo. Come

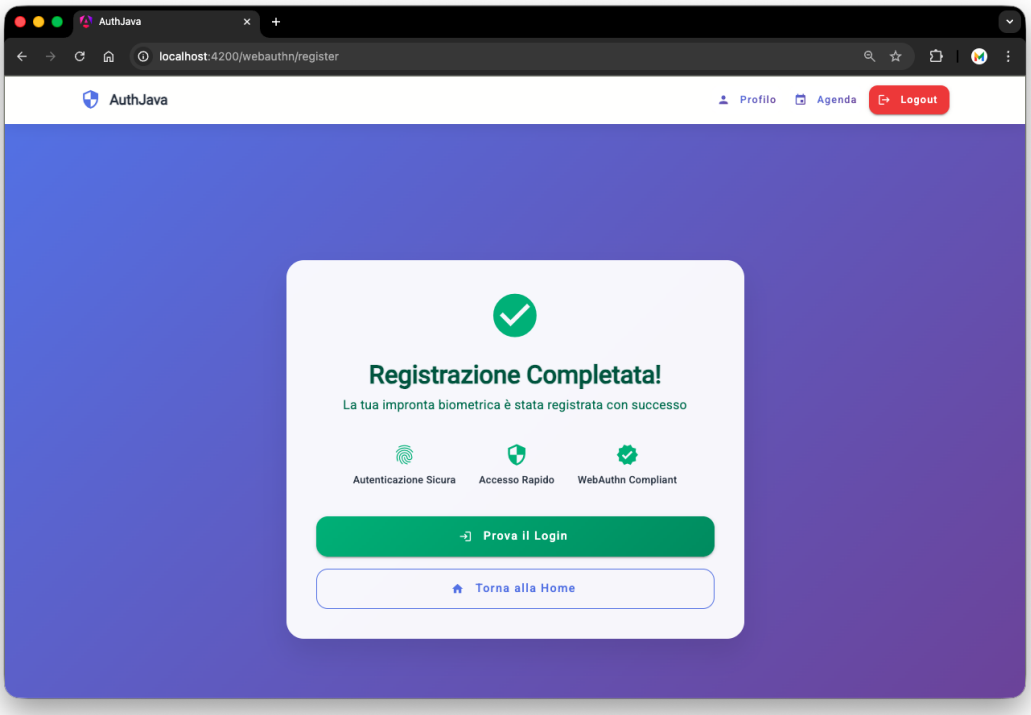

Figura 6.4: Registrazione impronta completata con successo

discusso nei capitoli precedenti, nelle altre schermate di login con inserimento di password, è necessario riconoscere le schermate esterne, come quella di Google, e ricordare la relativa password; difatti, ho avuto anch'io problemi a ricordarla. Inoltre, durante i processi di test, si sono verificate ulteriori situazioni di maggiore verifica da parte di Google, come quella di possesso. Infatti, Google chiede di possedere uno smartphone per poter verificare ulteriormente l'identità, aprendo un'app di proprietà di Google (come YouTube) e cliccando su un numero per poter continuare. Un processo che si può evitare grazie alle API suggerite. In questo modo, il login del browser avviene direttamente nel browser, senza dover installare applicazioni esterne o riconoscere schermate esterne, aiutando l'utente a percepire l'esperienza come più nativa.

6.4.3 Gestione dei casi di fallback

Se il dispositivo non dispone di un sensore biometrico o se l'utente non desidera utilizzarlo, può sempre configurare un metodo alternativo. Lo standard delle passkey, infatti, permette di sostituire l'impronta digitale o la caratteristica biometrica con

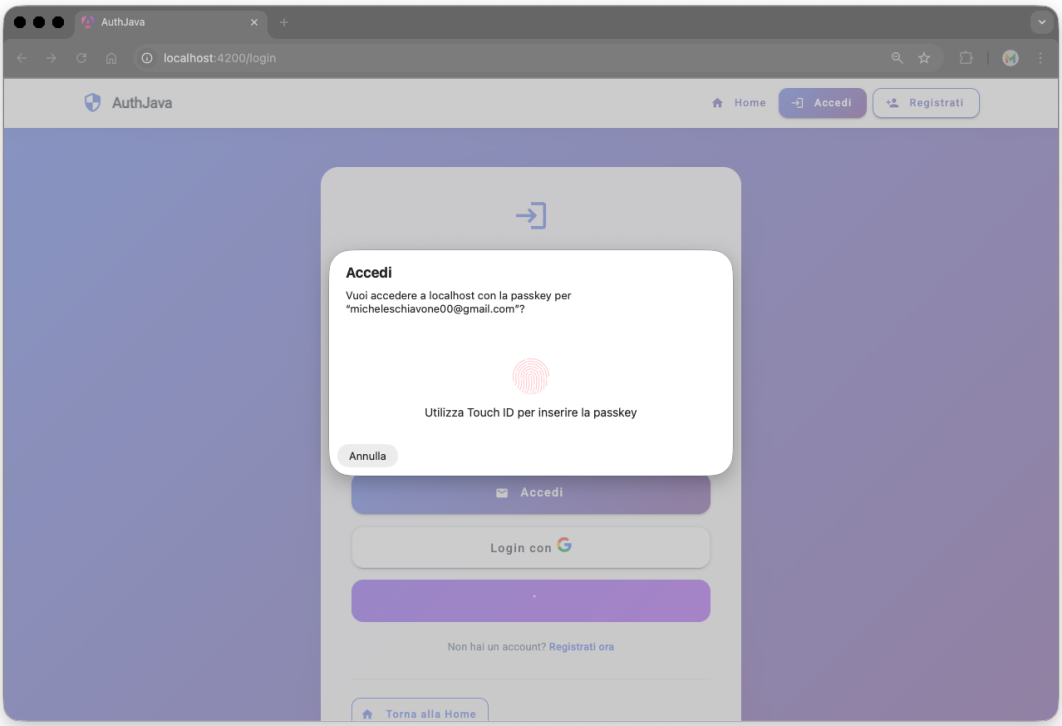

Figura 6.5: Modale di sistema per l'accesso tramite impronta

un PIN a 6 cifre o di scegliere di configurare un dispositivo esterno. In questo caso, però, si incorre nei problemi legati alle password, ovvero bisogna ricordare il PIN associato al sito web. Nel secondo caso, invece, viene mostrato un codice QR (Fig. 6.6)da scannerizzare con il dispositivo esterno (Fig. 6.7)e utilizzare il sensore presente su quel dispositivo per effettuare l'accesso.

6.5 Validazione del flusso di login e gestione delle sessioni

Per assicurarsi che il sistema funzioni correttamente, sono stati eseguiti dei test che coprono diverse aree: casi funzionali, come il login e il logout corretti o falliti, e il refresh; casi per verificare la gestione corretta dei token, un elemento cruciale, come token scaduti o assenti; casi per verificare la protezione delle API, che, nonostante siano semplici in questo caso, rappresentano una risorsa privata in caso di futuri utilizzi del proxy con casi più complessi.
Per testare il corretto funzionamento, è stato attivato un logging sia lato proxy che

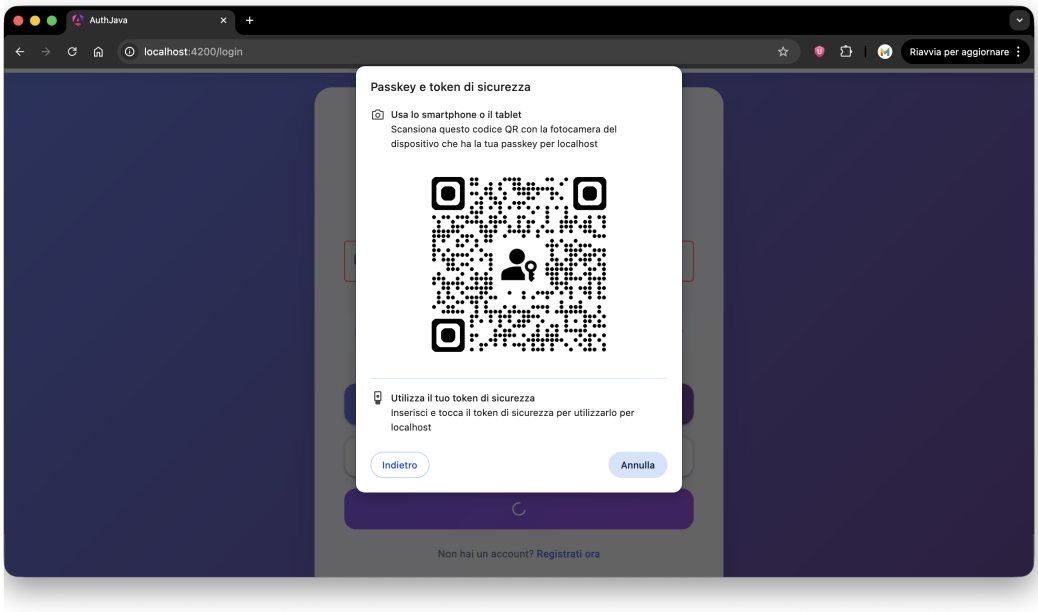

Figura 6.6: Utilizzo delle passkey come alternativa al sensore biometrico

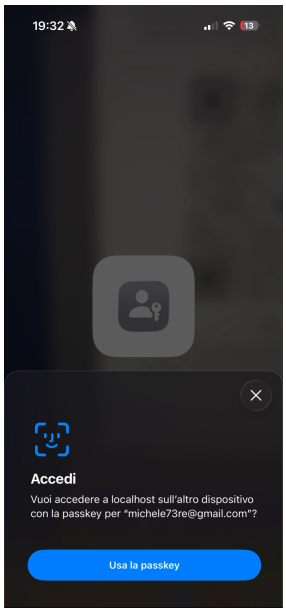

Figura 6.7: Visualizzazione dal dispositivo esterno per utilizzare il sensore del dispositivo

lato backend per verificare e provare i vari risultati in relazione alle diverse casistiche e per capire immediatamente, tramite il comportamento delle varie schermate, cosa sia andato storto.

6.5.1 Casi di test funzionali

Login corretto (happy path)

Il primo caso testato è ovviamente quello di verificare la registrazione e il corretto login con WebAuthn e il conseguente ottenimento dei token, come quello mostrato fin ora nel corso di questo capitolo.
Seguendo i vari passaggi, durante la fase di registrazione, il sistema dovrebbe consentire all'utente di accedere tramite google, ricevere i dati da quest'ultimo e verificare che si tratti effettivamente del primo login, criptare il jwt e impostarlo come cookie per il frontend. Siccome poi il sistema rimandi alla area privata in cui è possibile registrare l'impronta, dovrebbe all'accesso verificare l'utente loggato nel sistema chiamando la GET a /auth/user.
Dai dati emersi nel Log (come mostrato nel Log 6.5.1) tutto questo avviene correttamente, i dettagli dell'utente vengono ricavati dalle api di google (come sub, email_verified, given_name, family_name), che vengono poi salvati nel database e associati all'utente. Viene successivamente rilevato che si tratti effettivamente del primo login in modo da condurlo a registrare l'impronta biometrica, rispettando anche il procedimento di crittografia del jwt. Aprendo i DevTools di Google Chrome si può notare come questo viene impostato come Cookie. Inoltre le successive GET a /auth/user restituiscono oggetti che fanno capire al proxy che un utente è effettivamente autenticato nel sistema.

```
=== DATI RICEVUTI DA GOOGLE ===
[PROXY] sub: 110594927431109034375
[PROXY] email_verified: true
[PROXY] name: Michele Schiavone
[PROXY] given_name: Michele
[PROXY] family_name: Schiavone
[PROXY] email: user_email@gmail.com
[PROXY] picture: https://lh3.googleusercontent.com/...
[PROXY] exp: 2025-10-18T10:51:24Z

=== DATI SALVATI NEL DATABASE ===
[PROXY] ID: 1
[PROXY] Email: user_email@gmail.com
[PROXY] Nome: Michele
[PROXY] Cognome: Schiavone
[PROXY] Google Sub: 110594927431109034375
[PROXY] Refresh Token: presente e crittografato (lunghezza 176)
[PROXY] Primo login: true
```

```
[PROXY] Numero credenziali WebAuthn: 0

[PROXY] JWT generato e crittografato → impostato come cookie nel frontend
[PROXY] Cookie impostato con ID Token di Google

=== DEBUG /auth/user ===
[PROXY] Richiesta autenticata con OAuth2AuthenticationToken
[PROXY] Principal: 110594927431109034375
[PROXY] Scopes: [OIDC_USER, SCOPE_email, SCOPE_profile, SCOPE_openid]
[PROXY] Stato: autenticazione riuscita
```

Log 6.5.1: Flusso corretto di autenticazione e registrazione iniziale

Durante la fase di registrazione dell'impronta lo scambio delle opzioni di registrazione avviene in modo corretto, permettendo di registrare la propria impronta e associarla al proprio account, come dimostrato nel Log 6.5.2

```
=== INIZIO REGISTRAZIONE WEBAuthn ===
[FRONTEND] Opzioni di registrazione ricevute:
  - Attestation: none
  - Resident Key: preferred
  - User Verification: preferred
  - RP ID: localhost
  - Utente: user_email@gmail.com

[FRONTEND] Esecuzione chiamata: navigator.credentials.create()

[FRONTEND] Credenziale generata con successo:
  - ID: o2ZVZGB0lOVU3ZepC5p4INDAB5s
  - Tipo: public-key
  - ClientDataJSON e attestationObject ricevuti correttamente

[FRONTEND] Registrazione completata con successo
```

Log 6.5.2: Registrazione biometrica completata correttamente

Una volta registrata la propria impronta, è possibile effettuare il login con questa. In questa fase, il frontend e il proxy si scambiano continuamente informazioni: il proxy genera la challenge per l'URL. Come si può vedere dal log 6.5.3, le challenge hanno lo stesso valore, ma sono rappresentate in due modi diversi: la prima è in esadecimale, la seconda in Base64. Se l'utente inserisce l'impronta corretta, il processo di accesso prosegue, viene effettuata la verifica e l'utente viene reindirizzato al sistema, con i cookie impostati correttamente.

```
=== DEBUG /authenticationOptions ===
[PROXY] Session ID: 037F200A8FA36F7719EA8D9AA593FADE
[PROXY] Challenge generato e salvato: 77DE07933A8146CF904F139AD8A27E6F
```

```
4
5  [FRONTEND] Opzioni di autenticazione ricevute:
6    - rpId: localhost
7    - userVerification: required
8    - allowCredentials: []
9    - challenge: d94HkzqBRs+QTxOa2KJ+bw==
10
11 === DEBUG WebAuthn Login ===
12 [PROXY] Request Body ricevuto:
13   id: o2ZVZGB0lOVU3ZepC5p4INDAB5s
14   type: public-key
15   response: {clientDataJSON, authenticatorData, signature, userHandle}
16
17 [FRONTEND] Autenticazione avvenuta con successo:
18   PublicKeyCredential → id: o2ZVZGB0lOVU3ZepC5p4INDAB5s
19   authenticatorAttachment: platform
20   type: public-key
21
22 [FRONTEND] Login WebAuthn completato con successo
23 [FRONTEND] Dati utente ricevuti:
24   - Email: user_email@gmail.com
25   - Nome: Michele Schiavone
26
27 === DEBUG: WebAuthn Authentication ===
28 [PROXY] Principal: Name: [1], Granted Authorities: [OIDC_USER, ROLE_USER]
29 [PROXY] Attributi utente: {name=Michele Schiavone,
   ↪  email=user_email@gmail.com}
30 [PROXY] ID Token rinnovato con successo (lunghezza: 1162)
31 [PROXY] JWT criptato e impostato come cookie nel frontend
32 [PROXY] Cookie impostato con ID Token di Google (token decrittografato)
```

Log 6.5.3: Login biometrico corretto tramite WebAuthn

Login fallito (biometria errata)

Testando invece il caso di login fallito, si osserva un duplice comportamento. Al momento di effettuare il login, la funzione del frontend che si occupa di recuperare le credenziali associate all'impronta, ovvero navigator.credentials.get(), attiva il browser affinché attenda l'impronta. L'esito di questa chiamata dipende dal comportamento dell'utente: se l'utente inserisce un'impronta errata, è il sistema a occuparsi di questa eventualità. Nel mio caso, infatti, il browser mostra un'animazione che indica che l'impronta non è associata nemmeno al dispositivo. Quindi il sistema rimane in attesa di un'impronta effettivamente associata al dispositivo, più che alla PWA. Se invece l'utente annulla il processo, il browser genera un errore, annulla la Promise e impedisce la trasmissione dei dati al backend. Sulla console

viene infatti generato un log come quello presente nel Log 6.5.4, comportamento fondamentale per preservare la sicurezza dell'intero processo.

```
=== Inizio login con TouchID ===
login.component.ts:69 Authentication Options received: {challenge: {...},
↪ timeout: null, rpId: 'localhost', allowCredentials: Array(0),
↪ userVerification: 'required', ...}
login.component.ts:70 Challenge structure: {value:
↪ 'qHmy1IeNRVKrMFx/a+Z5LQ==’}
login.component.ts:125 Errore durante il login TouchID: NotAllowedError: The
↪ operation either timed out or was not allowed
```

Log 6.5.4: Fallimento del processo di autenticazione biometrica

Logout e revoca

Durante la fase di logout ci si aspetta che i token locali vengano cancellati; l'obiettivo di questo test è pertanto verificare che la procedura di logout invalidi correttamente la sessione dell'utente, impedendogli di utilizzare i token emessi in precedenza. Ciò avviene correttamente (Log 6.5.5), infatti, dopo l'operazione di logout, viene effettuato un altro test accedendo all'endpoint protetto utilizzando un vecchio JWT. Dopo la revoca, qualsiasi richiesta produce l'errore visualizzato nel log 6.5.6, confermando che la sessione precedente è stata invalidata correttamente.

```
[PROXY] === DEBUG /auth/logout ===
[PROXY] JWT revocato con successo: 936639cb1dfd5e35ebe77b5d136c5616
[PROXY] Session invalidated
[PROXY] Security context cleared
```

Log 6.5.5: Logout e invalidazione della sessione

```
401 Unauthorized - Token revocato. Rieffettuare il login.
```

Log 6.5.6: Errore di JWT non valido

6.5.2 Protezione delle API

La protezione delle API avviene internamente nel proxy sviluppato a tale scopo. A questo fine è stato realizzato un controller aggiuntivo denominato BackendProxy-Controller che svolge il ruolo di Security Gateway e si posiziona tra il frontend e il backend. La sua funzione principale è infatti quella di intercettare tutte le richieste provenienti dal frontend, validare i token JWT e infine inoltrare le richieste al backend dopo averle autenticate e autorizzate. Il caso di test precedente relativo

al token revocato è stato verificato proprio in questo controller, perché il frontend non invia le richieste direttamente al backend, ma all'endpoint /api/proxy/* che si occupa principalmente dell'inoltro delle richieste. In questo gateway, il token viene estratto dal cookie (id_token = <token>), viene verificato che sia stato criptato dal frontend e viene inoltrato al backend. Prima di essere inoltrato, il JWT subisce ulteriori validazioni: viene estratto un ID univoco dal JWT e si verifica che non sia stato revocato in precedenza dopo il logout; in tal caso, viene restituito l'errore 401. Dopo aver verificato anche la scadenza, se supera questi controlli, il proxy lo firma per garantire al backend che si tratti di una richiesta proveniente da un proxy autorizzato. A questo punto, il proxy inoltra la richiesta al backend e attende la risposta, che provvede a ritornare al frontend senza modifiche. Strutturata in questo modo, l'architettura presenta numerosi vantaggi: innanzitutto, tutta la validazione del JWT avviene nel proxy, lasciando al backend solo la verifica dell'autenticità dei certificati con i servizi Google, rappresentando un Single Point of Control. In questo modo, il backend risulta protetto, perché riceve soltanto richieste pre-autenticate e le responsabilità sono correttamente separate, lasciando al backend solo il compito di concentrarsi sulla logica di business.

Accesso con token valido

L'obiettivo di questo test è verificare che un utente autenticato possa accedere ai propri appuntamenti con un JWT valido. Chiamando la GET su api/proxy/appointments con l'header corretto, ci si aspetta di ottenere una risposta HTTP 200 OK contenente l'elenco degli appuntamenti che il frontend si occuperà di visualizzare. Il risultato osservato è riportato nel Log. 6.5.7.


```
[
    {
        "id": 1,
        "title": "Riunione di progetto",
        "description": "Discussione sui progressi del progetto di tesi",
        "dateTime": "2025-10-26T10:16:01.304019",
        "location": "Aula 202",
        "status": "SCHEDULED",
        "userEmail": "user_email@gmail.com"
    },
    {
        "id": 2,
        "title": "Sessione di studio",
        "description": "Preparazione esame finale",
        "dateTime": "2025-10-28T10:16:01.304041",
        "location": "Biblioteca",
        "status": "SCHEDULED",
        "userEmail": "user_email@gmail.com"
    }
```

```
]
```

Log 6.5.7: Lista di appuntamenti correttamente ricevuti

Accesso senza header di autorizzazione

Con questo test si vuole verificare che l'endpoint rifiuti le richieste prive di token. A tale scopo, la richiesta è stata configurata appositamente senza l'header Cookie, aspettandoci una risposta HTTP 401 Unauthorized. Il risultato osservato è riportato nel log. 6.5.8.

```
401 Unauthorized: Token non trovato.
```

Log 6.5.8: Risposta ricevuta a seguito di una richiesta senza cookie

Token scaduto o manomesso

Il token può scadere o essere manomesso prima di essere inoltrato al proxy. In tal caso, ci si aspetta che la verifica non vada a buon fine e che il proxy rifiuti la richiesta con una risposta HTTP 401 (Unauthorized) e fornisca un messaggio che indichi che il token è scaduto o non valido. A questo scopo è stato utilizzato un JWT scaduto o con una firma alterata e il risultato è riportato nel log. 6.5.9 e 6.5.10

```
401 Unauthorized: JWT criptato non valido. Rieffettuare il login.
```

Log 6.5.9: Risposta ricevuta a seguito di una richiesta con un JWT alterato

```
401 Unauthorized: Token scaduto. Rieffettuare il login.
```

Log 6.5.10: Risposta ricevuta a seguito di una richiesta con un JWT scaduto

6.6 Analisi del traffico e stress test

Per analizzare il traffico dell'app e simulare il suo utilizzo effettivo da parte di più utenti, sono stati effettuati dei test specifici con Apache Jmeter, configurando diversi gruppi di thread per simulare utenti concorrenti.
Per questo tipo di PWA è difficile testare l'effettivo login o la registrazione delle impronte, poiché il procedimento richiede un sensore biometrico che, nel funzionamento delle API, prevede la firma della challenge quando si richiede al server

tramite gli endpoint /api/webauthn/attestationOptions e /api/webauthn/authenticationOptions. Per questo motivo, sono state testate in questa modalità soltanto le richieste GET a questi endpoint, a /api/proxy/appointments, che permette di ricevere gli appuntamenti previa verifica del proxy, e /auth/user, che permette di ricevere i dati dell'utente loggato al sistema.
I test effettuati sono di quattro tipi diversi: test di base, per verificare il corretto funzionamento con carichi minimi; test di stress, per valutare la resistenza a carichi elevati; test di picco, per osservare la reazione a picchi improvvisi di richieste; test di stabilità, per analizzare la stabilità del sistema nel tempo.

6.6.1 Test Baseline

Il test baseline (di base) è stato condotto per verificare che il sistema abbia un comportamento corretto in condizioni di carico minimo. Sono state inviate 30 richieste per ciascuno dei 4 endpoint principali. Tutti gli endpoint hanno risposto correttamente, senza errori, con una media di risposta nell'ordine dei millisecondi. Si osserva, però, che l'endpoint /appointments ha un tempo di risposta più lungo, dovuto a logiche di verifica del JWT più complesse nel proxy e all'accesso a database e servizi esterni. Il throughput è simile per tutti gli endpoint, circa 6 richieste al secondo, il che mostra che il sistema gestisce correttamente il parallelismo, come mostrato nella tabella in Figura 6.8.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
attestationOptions	30	4	1	26	4.40	0.000%	6.37755	5.34	15.98	857.0
authenticationOptions	30	1	0	3	0.87	0.000%	6.41163	4.26	0.94	680.0
appointments	30	83	54	209	26.16	0.000%	6.31446	5.56	15.93	902.0
auth/user	30	2	1	10	1.61	0.000%	6.60066	3.53	16.40	547.0
TOTAL	120	22	0	209	37.31	0.000%	25.08886	18.29	48.04	746.5

Figura 6.8: Risultati del test baseline

6.6.2 Stress test

Lo stress test, è stato condotto con 5000 richieste per endpoint, per un totale di 20000 campioni, il che rappresenta un carico 167 volte superiore al test base, al fine di valutare se il sistema ha la capacità di sostenere un numero elevato di traffico simultaneo.
Dalla Tabella in Figura 6.9 si può notare come i tempi medi di risposta aumentano in modo significativo rispetto al test di base, con valori più elevati per l'endpoint di richiesta verso il backend, che rappresenta di nuovo l'operazione più onerosa. Da notare come il throughput complessivo si mantenga stabile e la percentuale di errore resti inferiore all'1%. Rispetto al test di base, si è verificato un singolo errore lato

server (codice 500), probabilmente dovuto all'elevato numero di richieste simultanee. Nonostante ciò, il sistema ha dimostrato una buona robustezza complessiva anche in condizioni di carico intenso, senza degradi significativi nelle prestazioni.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
attestationOptions	5000	1	0	269	5.78	0.000%	83.09098	69.54	208.21	857.0
authenticationOptions	5000	0	0	36	1.35	0.000%	83.11308	55.19	12.17	680.0
appointments	5000	68	35	377	32.44	0.120%	83.03717	73.24	209.21	903.1
auth/user	5000	1	0	58	2.43	0.000%	83.40284	44.55	207.20	547.0
TOTAL	20000	17	0	377	33.41	0.030%	332.03838	242.15	635.53	746.8

Figura 6.9: Risultati dello stress test

6.6.3 Test di picco (Spike test)

È stato condotto anche un test di picco per valutare se un aumento improvviso del numero di utenti potesse compromettere il sistema. Il numero di utenti è stato aumentato a 500 in pochi secondi, ma come mostrato nella tabella in figura 6.10, non si sono riscontrati problemi. Infatti, il sistema ha mantenuto un throughput complessivo di circa 396 richieste al secondo, un valore molto elevato rispetto alla complessità dei test, e ha mantenuto una percentuale di errore pari a 0%. Si nota inoltre che il tempo medio di risposta complessivo (Average nella tabella) è di circa 16 ms, con valori massimi pari a 65 ms registrati durante le richieste più onerose per l'endpoint di inoltro al server, sempre a causa delle operazioni più complesse. Questo test ha quindi dimostrato che il sistema reagisce correttamente anche in presenza di picchi improvvisi di traffico, mostrando una buona scalabilità orizzontale e l'assenza di degrado delle prestazioni.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
attestationOptions	500	0	0	18	1.12	0.000%	100.12014	83.79	250.89	857.0
authenticationOptions	500	0	0	3	0.48	0.000%	100.48232	66.73	14.72	680.0
appointments	500	65	34	282	35.66	0.000%	99.44312	87.60	250.84	902.0
auth/user	500	0	0	13	0.79	0.000%	104.03662	55.57	258.47	547.0
TOTAL	2000	16	0	282	33.14	0.000%	396.11804	288.77	758.48	746.5

Figura 6.10: Risultati del test di picco

6.6.4 Test di stabilità (Soak test)

Il test di stabilità è stato invece eseguito per una durata più prolungata (quasi 30 minuti), mantenendo un carico costante di 300 utenti. L'obiettivo di questo test è verificare la capacità del sistema di sostenere nel tempo un carico di richieste senza

compromettere le prestazioni.
Dai risultati, visibili in Figura 6.11, emerge un throughput complessivo di circa 680 richieste al secondo e una percentuale di errore trascurabile (solo lo 0,003%). Il tempo medio di risposta complessivo si attesta invece intorno ai 394 ms, con deviazioni più ampie per gli endpoint più onerosi. Si nota infatti che gli endpoint di webauthn, attestationOptions, authenticationOptions e auth/user mantengono tempi medi inferiori a 150 ms, risultando stabili e regolari anche nel lungo periodo. L'endpoint di inoltro al server, invece, presenta una maggiore variabilità, con un tempo medio di circa 1.169 ms e picchi fino a 648 s (valore anomalo dovuto a timeout isolati, probabilmente causati da ritardi nel backend o nella rete).
L'analisi del grafico dei tempi di risposta, mostrato in Figura 6.12, mostra che le richieste si sono mantenute al di sotto dei 300 ms, mentre /appointments ha oscillato tra i 300 ms e i 2.400 ms, raggiungendo un picco massimo di circa 5.100 ms. Possiamo quindi affermare che il sistema nel complesso ha mostrato una buona resistenza nel tempo, con prestazioni stabili e assenza di fenomeni di degradazione progressiva.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
attestationOptions	165558	128	0	1890	128.16	0.000%	183.92411	153.92	460.89	857.0
authenticationOptions	165539	129	0	1872	131.09	0.000%	183.90974	122.12	26.94	680.0
appointments	165504	1169	45	648855	3475.26	0.011%	169.96280	149.72	428.72	902.0
auth/user	165291	149	0	1891	137.58	0.000%	183.72056	98.02	456.43	546.3
TOTAL	661892	394	0	648855	1798.17	0.003%	679.70224	495.44	1301.34	746.4

Figura 6.11: Risultati del test di stabilità

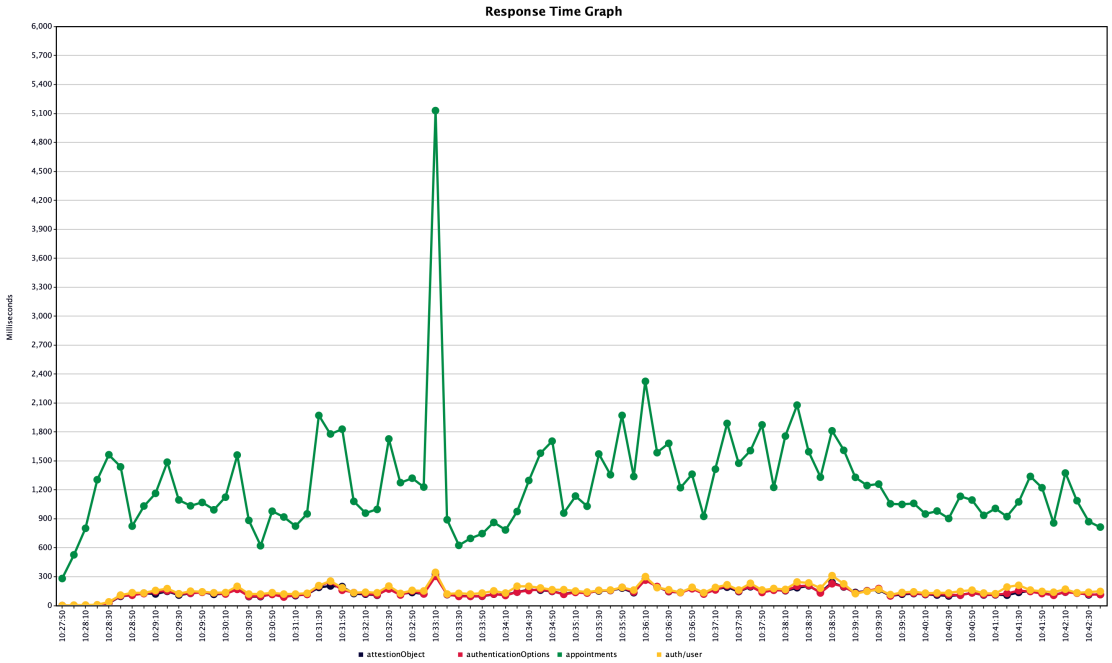

Figura 6.12: Response time graph del test di stabilità

6.6.5 Analisi complessiva

Nel complesso, il sistema si è dimostrato stabile, scalabile e privo di gravi errori. Gli unici errori si sono verificati durante lo stress test e il test di stabilità, casi eccezionali che hanno comunque mantenuto la percentuale di errore complessiva sotto lo 0,6%. Il throughput si è rivelato costante e la capacità di recupero efficace sotto carico prolungato.

Test	Endpoint	Average (ms)	Throughput (req/s)	Error (%)
Baseline	attestationOptions	4	6.36	0.000%
Baseline	authenticationOptions	1	6.40	0.000%
Baseline	appointments	63	6.31	0.000%
Baseline	auth/user	2	6.6	0.000%
Stress	attestationOptions	1	83.09	0.000%
Stress	authenticationOptions	0	83.11	0.000%
Stress	appointments	68	83.03	0.120%
Stress	auth/user	1	83.4	0.000%
Spike	attestationOptions	0	100.12	0.000%
Spike	authenticationOptions	0	100.48	0.000%
Spike	appointments	65	99.44	0.000%
Spike	auth/user	0	104.03	0.000%
Soak	attestationOptions	128	183.92	0.000%
Soak	authenticationOptions	129	183.91	0.000%
Soak	appointments	1169	169.96	0.011%
Soak	auth/user	149	183.72	0.000%

Figura 6.13: Analisi complessiva - Throughput, tempo medio e percentuale di errore

Capitolo 7

Conclusioni

Come dimostrato dai test, l'autenticazione basata sulle API WebAuthn, sfruttata dal proxy, è una soluzione sicura ed efficiente. I test di picco e i test di stress hanno inoltre dimostrato la sua scalabilità, rendendolo adatto alla gestione dell'accesso degli utenti. Il software utilizzato, Apache JMeter, e le sue misurazioni hanno evidenziato prestazioni complessivamente stabili: i tempi medi di risposta sono contenuti e il throughput si è dimostrato costante anche in presenza di carichi elevati. Non sono mancati però rallentamenti legati a operazioni più complicate, che però non hanno compromesso la stabilità complessiva del sistema, confermandone la robustezza e l'affidabilità, criteri molto importanti per la sicurezza e l'autenticazione, soprattutto in scenari di utilizzo intensivo.
Le principali criticità riscontrate durante la realizzazione del progetto sono state legate all'integrazione con il componente OAuth2-Proxy, come già menzionato più volte nel corso dell'elaborato. In particolare, per garantire una comunicazione sicura tra il proxy sviluppato e la PWA di destinazione, è stata necessaria un'attenta configurazione della gestione dei token e della modalità di propagazione delle richieste autenticate. Un esempio è la scelta della revoca dei JWT, basata su black list piuttosto che altre soluzioni, forse la meno scalabile, ma giustificata dal contesto e in questo caso efficiente. Queste difficoltà hanno rappresentato una sfida significativa, ma hanno permesso di acquisire una comprensione più approfondita delle dinamiche di autenticazione e dei meccanismi di sicurezza associati.
Poiché il proxy risulta avere un'elevata adattabilità, in futuro potrà essere facilmente integrato in applicazioni web già esistenti. Infatti, può essere inserito come mediatore nella comunicazione tra client e server, delegandogli la gestione delle operazioni di autenticazione. Tra i possibili miglioramenti, vi sono l'ottimizzazione delle richieste verso il backend con una logica meno complessa ma sicura, l'ottimizzazione del caching e della gestione delle sessioni per ridurre ulteriormente la latenza lato proxy.

Bibliografia

[1] Andreas Biørn-Hansen, Tim A. Majchrzak e Tor-Morten Grønli. «Progressive Web Apps: The Possible Web-native Unifier for Mobile Development». In: (2017), pp. 344–351. DOI: 10.5220/0006353703440351 (cit. a p. 4).

[2] David Fortunato e Jorge Bernardino. «Progressive web apps: An alternative to the native mobile Apps». In: *2018 13th Iberian Conference on Information Systems and Technologies (CISTI)* (2018), pp. 1–6. DOI: 10.23919/CISTI.2018.8399228 (cit. alle pp. 4, 8, 18).

[3] Dean Alan Hume. «Progressive Web Apps». In: (2017) (cit. a p. 5).

[4] Kashish Behl e G. Raj. «Architectural Pattern of Progressive Web and Background Synchronization». In: *2018 International Conference on Advances in Computing and Communication Engineering (ICACCE)* (2018), pp. 366–371. DOI: 10.1109/ICACCE.2018.8441701 (cit. alle pp. 5–8).

[5] Pratik Thacker e Dr.Andhe Dharani. «Realization of native apps using Progressive Web Apps». In: *Journal of emerging technologies and innovative research* (2020) (cit. alle pp. 6, 12).

[6] Varsha Sharma, Rajat Verma, Vaishali Pathak, Muskan Paliwal e P. Jain. «Progressive Web App (PWA) - One Stop Solution for All Application Development Across All Platforms». In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* (2019). DOI: 10.32628/CSEIT1952290 (cit. alle pp. 6, 7).

[7] Bangar Raju Cherukuri. «Progressive Web Apps (PWAs): Enhancing User Experience through Modern Web Development». In: *International Journal of Science and Research (IJSR)* (2024). DOI: 10.21275/ms241022095359 (cit. alle pp. 7, 8).

[8] Peter Gasston. «The Modern Web: Multi-Device Web Development with HTML5, CSS3, and JavaScript». In: (2013). DOI: 10.5860/choice.51-0928 (cit. a p. 8).

[9] MDN Web Docs. *MediaDevices.getUserMedia()*. https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia. [Online; accessed 9-July-2025] (cit. a p. 9).

[10] SimiCart. *How to Access Camera in PWA*. https://simicart.com/blog/pwa-camera-access/ (cit. a p. 9).

[11] MDN Web Docs. *Web Storage API*. https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API. [Online; accessed 19-July-2025] (cit. a p. 11).

[12] MDN Web Docs. *Web Bluetooth API*. https://developer.mozilla.org/en-US/docs/Web/API/Web_Bluetooth_API. [Online; accessed 19-July-2025] (cit. a p. 13).

[13] MDN Web Docs. *WebUSB API*. https://developer.mozilla.org/en-US/docs/Web/API/WebUSB_API. [Online; accessed 19-July-2025; last modified Oct 8, 2024] (cit. a p. 14).

[14] Chrome Developers. *Interact with NFC devices on Chrome for Android (Web NFC)*. https://developer.chrome.com/docs/capabilities/nfc. [Online; accessed 19-July-2025; last updated 12 February 2020] (cit. a p. 15).

[15] MDN Web Docs. *Payment Request API*. https://developer.mozilla.org/en-US/docs/Web/API/Payment_Request_API. [Online; accessed 2025-08-19; last modified 10 April 2025] (cit. a p. 16).

[16] MDN Web Docs. *Web Share API*. https://developer.mozilla.org/en-US/docs/Web/API/Web_Share_API. [Online; accessed 2025-07-19; last modified 13 March 2025] (cit. a p. 16).

[17] MDN Web Docs. *Screen Wake Lock API*. https://developer.mozilla.org/en-US/docs/Web/API/Screen_Wake_Lock_API. [Online; accessed 2025-07-19; last modified 3 July 2025] (cit. a p. 17).

[18] Drew Jabin. *The True Costs of Password Management*. https://conduitstreet.mdcounties.org/2020/03/24/the-true-costs-of-password-management/. [Online; accessed 21-August-2025; published 24 March 2020] (cit. a p. 21).

[19] Fran Rosch. *Embracing The End Of The Password Here And Now*. https://www.forbes.com/councils/forbestechcouncil/2023/03/23/embracing-the-end-of-the-password-here-and-now/. [Online; accessed 21-August-2025; published 23 March 2023] (cit. a p. 22).

[20] Andrés Aguiar. *What Is Passwordless Authentication?* https://auth0.com/blog/what-is-passwordless-authentication/. [Online; accessed 07-August-2025; published 17 June 2021] (cit. alle pp. 22, 24).

[21] Deepu K Sasidharan. *A Passwordless Future: Passkeys for Developers*. https://auth0.com/blog/webauthn-and-passkeys-for-developers/. [Online; accessed 21-August-2025; published 14 February 2024] (cit. a p. 22).

[22] Chun-Ying Huang, Shang-Pin Ma e Kuan-Ta Chen. «Using one-time passwords to prevent password phishing attacks». In: *J. Netw. Comput. Appl.* 34 (2011), pp. 1292–1301. DOI: 10.1016/j.jnca.2011.02.004 (cit. a p. 22).

[23] Anargyros Chrysanthou, Yorgos Pantis e Constantinos Patsakis. «The anatomy of deception: Measuring technical and human factors of a large-scale phishing campaign». In: *Comput. Secur.* 140 (2024), p. 103780. DOI: 10.1016/j.cose.2024.103780 (cit. a p. 22).

[24] Alphiya Yunoose, Aby Rose Varghese, Anagha R, Abhirami Prakash e Devika Babu. «PHISHING». In: *international journal of engineering technology and management sciences* (2022). DOI: 10.46647/ijetms.2022.v06i05.092 (cit. a p. 22).

[25] Minh Hieu Nguyen Ba, Jacob Bennett, Michael Gallagher e S. Bhunia. «A Case Study of Credential Stuffing Attack: Canva Data Breach». In: *2021 International Conference on Computational Science and Computational Intelligence (CSCI)* (2021), pp. 735–740. DOI: 10.1109/CSCI54926.2021.00187 (cit. a p. 23).

[26] Edim Bassey Edim e Akpan Itoro Udofot. «Biometric Authentication and Algorithm: A review». In: *International Journal of Science and Research Archive* (2025). DOI: 10.30574/ijsra.2025.14.3.0473 (cit. alle pp. 25–28).

[27] WebAuthn4J Project. *WebAuthn4J Reference (version 0.29.6-SNAPSHOT)*. https://webauthn4j.github.io/webauthn4j/en/. [Online; accessed 18-August-2025; version 0.29.6-SNAPSHOT] (cit. alle pp. 28, 29, 31).

[28] Chrome for Developers. *What's new with sign up and sign in on the web (Google I/O '18)*. https://www.youtube.com/watch?v=kGGMgEfSzMw. [Online; accessed 18-August-2025] (cit. alle pp. 28–31).

[29] Dick Hardt. «The OAuth 2.0 Authorization Framework». In: *RFC* 6749 (2012), pp. 1–76. URL: https://api.semanticscholar.org/CorpusID:167313730 (cit. alle pp. 32–34).

[30] Michael B. Jones, J. Bradley e N. Sakimura. «JSON Web Token (JWT)». In: *RFC* 7519 (2015), pp. 1–30. DOI: 10.17487/RFC7519 (cit. alle pp. 34, 35).

[31] Sebastián Peyrott. *JWT Handbook*. [Online; accessed 21-August-2025; available via Internet Archive]. Auth0, 2018. URL: https://archive.org/details/auth0-jwt-handbook (cit. alle pp. 35, 39).

[32] OAuth2 Proxy Project. *Welcome — OAuth2 Proxy Documentation (v7.12.x)*. `https://oauth2-proxy.github.io/oauth2-proxy/`. [Online; accessed 21-August-2025; version 7.12.x] (cit. a p. 41).

[33] Brian Demers. *Add Auth to Any App with OAuth2 Proxy*. `https://developer.okta.com/blog/2022/07/14/add-auth-to-any-app-with-oauth2-proxy`. [Online; accessed 21-August-2025; published 14 July 2022] (cit. a p. 41).

[34] Ayan Chatterjee e A. Prinz. «Applying Spring Security Framework with KeyCloak-Based OAuth2 to Protect Microservice Architecture APIs: A Case Study». In: *Sensors (Basel, Switzerland)* 22 (2022). DOI: `10.3390/s22051703` (cit. a p. 43).

[35] Swarag Sharma e Jevitha Kp. «Security Analysis of OAuth 2.0 Implementation». In: *2023 Innovations in Power and Advanced Computing Technologies (i-PACT)* (2023), pp. 1–8. DOI: `10.1109/i-PACT58649.2023.10434479` (cit. alle pp. 44, 52).

[36] Srivathsan G. Morkonda, P. V. Oorschot e Sonia Chiasson. «Exploring Privacy Implications in OAuth Deployments». In: *ArXiv* abs/2103.02579 (2021) (cit. alle pp. 44, 45, 48, 49).

[37] Jaimandeep Singh e N. Chaudhary. «Unified Singular Protocol Flow for OAuth (USPFO) Ecosystem». In: *ArXiv* abs/2301.12496 (2023). DOI: `10.48550/arXiv.2301.12496` (cit. a p. 50).

[38] Tommaso Innocenti, Matteo Golinelli, Kaan Onarlioglu, Ali Mirheidari, Bruno Crispo e E. Kirda. «OAuth 2.0 Redirect URI Validation Falls Short, Literally». In: *Proceedings of the 39th Annual Computer Security Applications Conference* (2023). DOI: `10.1145/3627106.3627140` (cit. a p. 50).

[39] J. Bradley, Brian Campbell e Michael B. Jones. «OAuth 2.0 DPoP for the Implicit Flow». In: (2000) (cit. a p. 51).

[40] Jaimandeep Singh e N. Chaudhary. «OAuth 2.0 : Architectural design augmentation for mitigation of common security vulnerabilities». In: *J. Inf. Secur. Appl.* 65 (2022), p. 103091. DOI: `10.1016/j.jisa.2021.103091` (cit. a p. 51).

[41] E. Arshad, Michele Benolli e B. Crispo. «Practical attacks on Login CSRF in OAuth». In: *Comput. Secur.* 121 (2022), p. 102859. DOI: `10.1016/j.cose.2022.102859` (cit. a p. 52).