



**UNIVERSITÀ
DEGLI STUDI
DI UDINE**

**Dipartimento di Scienze
Matematiche, Informatiche e Fisiche**

TESI DI LAUREA IN
INFORMATICA

Progettazione e sviluppo di un servizio di autenticazione a due fattori

CANDIDATO

Lorenzo D'Antoni

RELATORE

Prof. Marino Miculan

TUTOR AZIENDALE

Federico Selatti

Anno accademico 2020-2021

CONTATTI DELL'ISTITUTO

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Università degli Studi di Udine

Via delle Scienze, 206

33100 Udine — Italia

+39 0432 558400

<https://www.dmif.uniud.it/>

Indice

1	Introduzione	1
1.1	Struttura della tesi	2
2	2FA: stato dell'arte	3
2.1	Introduzione	3
2.2	Funzionamento	4
2.3	Principali tipologie di 2FA	5
2.3.1	Token hardware	5
2.3.2	SMS Text-Message & Voice-based	7
2.3.3	Token Software	7
2.3.4	Push notification	8
2.3.5	Riconoscimento biometrico	8
2.4	Algoritmi utilizzati nel 2FA	8
2.4.1	HMAC	9
2.4.2	HOTP	10
2.4.3	TOTP	11
3	Analisi del problema	15
3.1	Requisiti	15
3.1.1	Requisiti funzionali	15
3.1.2	Requisiti non funzionali	16
4	Progettazione della soluzione	19
4.1	Selezione della modalità di 2FA	19
4.2	Scelta del bundle 2FA	20
4.2.1	Perché utilizzare una libreria?	20
4.2.2	Analisi dei bundle 2FA presenti sul mercato	21
4.3	Analisi delle funzionalità del servizio	22
4.4	Diagramma dei casi d'uso	23
4.5	Ricerca e analisi dei componenti del servizio	25
4.5.1	SQL vs NoSQL	26
4.5.2	Eliminazione automatizzata dei dati in cache	27
4.6	Attivazione e disattivazione del 2FA	30
4.6.1	Attivazione	30
4.6.2	Disattivazione	32
4.7	Dispositivi mobile	32
4.7.1	Web API	33
4.7.2	SOAP	34
4.7.3	REST	35
4.7.4	REST vs SOAP	36
4.8	Diagramma dei componenti	37

5	Implementazione	41
5.1	Tecnologie utilizzate	41
5.1.1	Scelta del servizio di <i>Cloud Storage</i>	42
5.2	Implementazione del servizio per browser web	43
5.2.1	Installazione del bundle 2FA	43
5.2.2	Attivazione del 2FA	46
5.2.3	Disattivazione del 2FA	56
5.2.4	Autenticazione tramite 2FA	56
5.3	REST API	58
5.3.1	Definizione dell'API	60
5.3.2	Implementazione dell'API	62
5.4	Creazione del bundle	64
6	Conclusioni	69
6.1	Uno sguardo al futuro	70

Elenco delle figure

2.1	I diversi fattori di autenticazione	4
2.2	Processo di autenticazione a due fattori basato su TOTP (da [12])	12
4.1	Diagramma dei casi d'uso (<i>Use Case Diagram</i>) raffigurante il servizio <i>2FA</i> da progettare	24
4.2	Schema raffigurante il funzionamento di <i>Google Cloud Pub/Sub</i> (da [4])	29
4.3	Diagramma di sequenza (<i>Sequence Diagram</i>) che illustra l'ordine delle interazioni necessarie all'attivazione del <i>2FA</i>	31
4.4	Diagramma delle classi (<i>class diagram</i>) raffigurante la struttura interna del servizio in fase di progettazione	39
5.1	Pagina web associata alla prima fase dell'attivazione del <i>2FA</i>	49
5.2	Autenticazione standard vs Autenticazione con 2FA (da [13])	58

1

Introduzione

Al giorno d'oggi l'autenticazione a due fattori (*2FA*) è uno dei sistemi più semplici ed efficaci per proteggere gli *account* degli utenti. Nonostante il suo utilizzo sia aumentato drasticamente negli ultimi anni, non è ancora uno standard adottato universalmente.

Come suggerisce il nome, il *2FA* introduce un secondo livello di sicurezza (fattore) al processo di *login*. Nella maggior parte dei casi, quest'ultimo avviene con la sola password: si tratta dunque di autenticazione ad un fattore (conoscenza).

Il *2FA*, invece, prevede l'utilizzo di due fattori: il primo, in genere, è quello legato alla conoscenza (una cosa che l'utente conosce) mentre il secondo è associato al possesso (una cosa che l'utente possiede) o all'inerenza (un dato biometrico che contraddistingue l'utente univocamente). Tuttavia, affinché si possa parlare di *2FA*, i fattori adottati devono essere di matrice differente: in questo modo è molto più difficile comprometterli entrambi. Una volta completata l'autenticazione tramite nome utente e password, le credenziali inserite (prima di raggiungere il *server*) attraversano la rete *internet*. È importante, quindi, che il secondo fattore utilizzi un canale diverso dal primo per evitare che la manomissione di uno dei due metta a repentaglio l'*account* dell'utente. Da ciò si desume che un'autenticazione basata solo su password, a prescindere dalla robustezza di quest'ultima, è intrinsecamente debole.

L'autenticazione a due fattori, pertanto, è una soluzione relativamente semplice che può incrementare notevolmente la sicurezza *online*. Il *2FA*, infatti, fornisce protezione contro i seguenti tipi di attacchi/minacce:

- **Password rubate.** Senza *2FA*, ogni persona che (in qualche modo) conosce la password può accedere all'*account* dell'utente
- **Social Engineering.** Spesso un attaccante cerca semplicemente di manipolare l'utente per conoscere le sue credenziali
- **Key Logging.** È possibile utilizzare un *malware* che registri tutti i tasti della tastiera premuti dall'utente
- **Tentativi di phishing.** Spesso vengono inviate *e-mail* che contengono *link* verso siti web gestiti da malintenzionati con lo scopo di convincere gli utenti a digitare le loro password o di compromettere i loro computer.

- **Attacchi Brute-Force.** Per determinare la password che permette di accedere ad un certo *account* vengono tentate tutte le possibili combinazioni di lettere, numeri e caratteri speciali oppure si utilizzano le credenziali che sono state sottratte nei precedenti attacchi informatici (*breach*)

Una ricerca [14] condotta da *Google*, dall'*Università di New York* e dall'*Università della California (San Diego)* ha evidenziato che il *2FA* basato sugli *SMS* (la più debole modalità di autenticazione a due fattori) protegge al 100% contro gli attacchi di *bot* automatizzati, al 96% contro gli attacchi di *phishing* massivi e al 76% contro attacchi mirati verso uno specifico utente.

Detto ciò, il *2FA* non elimina tutti i rischi ma sicuramente è un'ottima soluzione per incrementare la sicurezza dei propri dati personali. Ci sono numerosi metodi che un attaccante può utilizzare per compromettere l'*account* di un utente con autenticazione a due fattori attiva. Per citarne uno, è possibile sfruttare i punti deboli delle procedure di recupero degli *account* per arginare l'autenticazione a due fattori. L'introduzione del *2FA*, infatti, ha aumentato il rischio di non riuscire più ad accedere al proprio *account* (es. telefono smarrito) e, per questo motivo, sono state introdotte nuove procedure di *recovery*.

1.1 Struttura della tesi

La seguente tesi è associata all'attività di tirocinio svolta all'interno dell'azienda *Cogito srl*. L'obiettivo è stato lo sviluppo di un servizio di autenticazione a due fattori da poter utilizzare all'interno di una piattaforma web. I capitoli che seguono descrivono l'intero processo di progettazione ed implementazione del servizio.

Il capitolo 2 fornisce al lettore una panoramica generale sul *2FA*, essenziale per acquisire alcune conoscenze preliminari che faciliteranno la comprensione dei capitoli successivi.

Il capitolo 3 analizza più a fondo il sistema da progettare, delineando i requisiti funzionali e non funzionali che quest'ultimo dovrà soddisfare.

Il capitolo 4 esamina la fase di progettazione, caratterizzata dalla definizione di uno schema architetturale contenente tutti i componenti che andranno a comporre il servizio.

Il capitolo 5, infine, si occupa della fase implementativa definendo le tecnologie utilizzate e l'implementazione dei principali componenti del servizio.

2

2FA: stato dell'arte

L'autenticazione a due (*Two-Factor Authentication*, *2FA*) o più fattori (*Multi-Factor Authentication*, *MFA*) viene definita a livello europeo nella *PSD2* (*Payment Services Directive 2*) [1] come:

*un'autenticazione basata sull'uso di due o più elementi, classificati nelle categorie della **conoscenza** (qualcosa che solo l'utente conosce), del **possesso** (qualcosa che solo l'utente possiede) e dell'**inerenza** (qualcosa che caratterizza l'utente), che sono indipendenti (in quanto la violazione di uno non compromette l'affidabilità degli altri) ed è concepita in modo tale da tutelare la riservatezza dei dati di autenticazione.*

2.1 Introduzione

L'autenticazione a due fattori, dunque, aggiunge un ulteriore livello di sicurezza nella verifica dell'identità dell'utente. Come accennato nella precedente definizione, l'autenticazione nei sistemi digitali può avvenire tramite tre metodi distinti:

- “Una cosa che sai” (**conoscenza**), per esempio una password o un *PIN* (*Personal Identification Number*)
- “Una cosa che hai” (**possesso**), per esempio un telefono, una carta di credito o un *token* di sicurezza (chiavetta che genera un codice di accesso temporaneo)
- “Una cosa che sei” (**inerenza**), come l'impronta digitale, il timbro vocale o un altro dato biometrico.

Un'autenticazione basata solo su password richiede un solo fattore per concedere l'accesso al sistema. In questo caso si parla di **autenticazione debole**. Quando, invece, viene utilizzato più di un fattore si parla di **autenticazione forte** (*strong authentication*). L'autenticazione a due fattori, pertanto, è un metodo di autenticazione forte che combina l'utilizzo di due fattori appartenenti a categorie diverse.

Esiste, peraltro, anche l'autenticazione a tre fattori (*3FA*). È molto meno usata e viene impiegata, per esempio, nello *SPID* (Sistema Pubblico di Identità Digitale) di livello 3. I tre fattori in questione sono la password, l'applicazione *PosteID* e il *PIN SPID 3*.

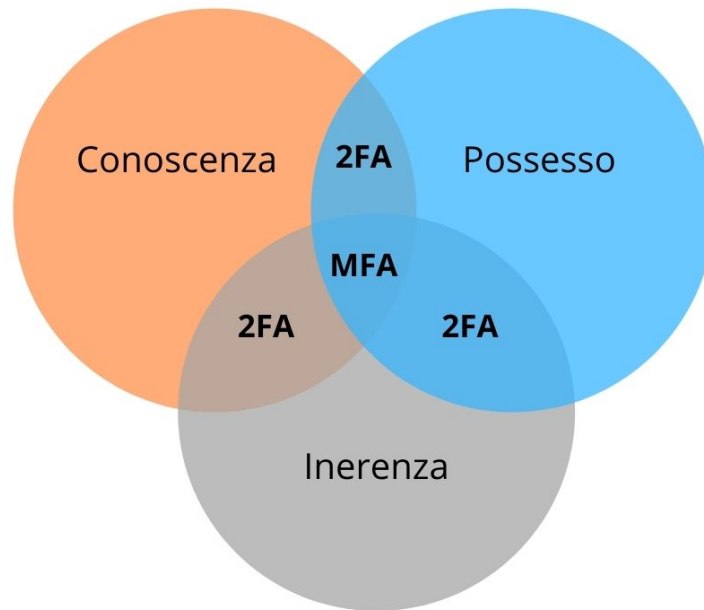


Figura 2.1: I diversi fattori di autenticazione

2.2 Funzionamento

Tipicamente, dopo aver inserito la password del proprio account (primo fattore), verrà richiesto di inserire un secondo fattore. In questo modo, anche se uno dei due verrà compromesso, l'account rimarrà al sicuro.

Un processo di autenticazione a due fattori è, ad esempio, lo sblocco dei telefoni di ultima generazione subito dopo l'accensione. L'utente, infatti, deve inserire il *PIN* (qualcosa che conosce) ed eseguire una scansione della propria impronta digitale o del proprio volto (qualcosa che lo caratterizza).

Il *NIST* (*National Institute of Standards and Technology*), un'agenzia del governo degli Stati Uniti d'America che ha il compito di sviluppare standard tecnologici, si occupa dell'uso delle password e dell'autenticazione a due fattori nella *SP* (*Special Publication*) 800-63, in particolare nella *SP 800-63-3* [7] e nella *SP 800-63B* [6].

Nella prima pubblicazione (*SP 800-63-3*), facendo riferimento al processo di autenticazione, vengono definiti i tre livelli *AAL* (*Authenticator Assurance Level*).

- **AAL1.** Richiede un'autenticazione ad un fattore o multifattoriale usando un protocollo di autenticazione sicuro.
- **AAL2.** Richiede di dimostrare il possesso di due distinti fattori di autenticazione tramite un protocollo di autenticazione sicuro.
- **AAL3.** Richiede di dimostrare il possesso di una chiave tramite un protocollo crittografico. Il *NIST* consiglia l'uso di un "autenticatore" hardware e di un dispositivo che non sia vulnerabile ad attacchi di tipo *man-in-the-middle* (*MitM*) e ad attacchi di *phishing*. Non è necessario avere due dispositivi separati se un dispositivo soddisfa entrambi i requisiti.

Il paragrafo 6.2 (*Selecting AAL*), invece, si occupa delle modalità con cui selezionare il livello *AAL* più appropriato per ogni tipo di servizio digitale.

Nella *SP 800-63B* vengono esaminate tutte le modalità di autenticazione a due fattori permesse in base al *AAL* richiesto. Viene specificato, inoltre, che il primo fattore deve essere basato sulla conoscenza (password) mentre il secondo deve essere basato sul possesso. Il dispositivo utilizzato come secondo fattore di autenticazione deve “garantire una resistenza all’impersonificazione”, cioè impedire l’uso a persone che non siano il legittimo proprietario.

Nel capitolo 5.1.3 vengono trattati i dispositivi *out-of-band*, ovvero dispositivi fisici indirizzabili in modo univoco e in grado di comunicare in modo sicuro con il sistema su cui l’utente sta effettuando l’autenticazione tramite un canale di comunicazione distinto (canale secondario). Questi dispositivi devono essere in possesso e sotto il controllo dell’utente.

Sempre in questo capitolo, viene considerato pericoloso l’utilizzo della linea telefonica per ricevere il codice di autenticazione. È consigliato, invece, l’utilizzo dei “*multi-factor OTP device*”: dispositivi hardware o generatori di *OTP* (*One Time Password*) installati, per esempio, sugli *smartphone*. Viene specificato, inoltre, che questi dispositivi devono essere preventivamente attivati da “qualcosa che sai” (password) o da “qualcosa che sei” (riconoscimento biometrico).

Nonostante gli standard definiti dal *NIST* non siano obbligatori in Europa, rimangono un punto di riferimento a livello mondiale per la loro completezza e autorevolezza. A livello europeo, in seguito alla Direttiva (UE) 2015/2366 nota come *PSD2* (*Payment Services Directive 2*) [1], l’autenticazione a due o più fattori, denominata *SCA* (*Strong Customer Authentication*), è stata ulteriormente rafforzata e resa obbligatoria per le operazioni bancarie. In base all’articolo 97, inoltre, l’uso della *SCA* è obbligatorio quando l’utente

1. accede al suo conto di pagamento online;
2. dispone un’operazione di pagamento elettronico;
3. effettua qualsiasi azione, tramite un canale a distanza, che può comportare un rischio di frode nei pagamenti o altri abusi.

Nell’articolo 98, infine, viene assegnato il compito di emanare le “Norme tecniche di regolamentazione in materia di autenticazione e comunicazione” a *EBA* (*European Banking Authority*). Quest’ultima ha rilasciato le *RTS* (*Regulatory Technical Standards*) che specificano anche i requisiti delle procedure di autenticazione forte e le relative esenzioni d’uso.

2.3 Principali tipologie di 2FA

Esistono diversi metodi con cui ottenere il secondo fattore di autenticazione. Alcuni risultano più sicuri e complessi ma tutti offrono una miglior protezione rispetto alla sola autenticazione tramite password.

2.3.1 Token hardware

Si tratta di dispositivi simili ad una chiavetta *USB* che forniscono nuovi *OTP* da inserire nei sistemi digitali per permettere l’autenticazione.

Si dividono in due tipi: **token disconnessi** e **token connessi** [8]. I primi fanno riferimento a dispositivi separati che non hanno una connessione diretta al sistema in uso. I token connessi, invece, trasmettono gli *OTP* al client tramite una connessione fisica, di solito per mezzo del *universal serial bus* (*USB*) a seguito della pressione di un pulsante. In passato venivano forniti dalla banche ma, in base alle ultime direttive europee, non sono considerati sicuri e sono in via di dismissione in quanto potevano essere attivati senza alcun codice di sicurezza.

In questi ultimi anni sono stati creati nuovi tipi di token che si basano sullo standard *FIDO U2F* (*Universal 2nd Factor*) *Security Key* [2]. Si tratta di uno standard di autenticazione *open-source* sviluppato inizialmente da *Google* e da *Yubico* e poi confluito nella *FIDO* (*Fast IDentity Online*) *Alliance*. Ogni dispositivo è dotato di una piccola quantità di memoria per salvare un certificato o un identificatore univoco necessario durante la procedura di autenticazione. I dispositivi più semplici richiedono l'inserimento in una porta *USB* e la comunicazione con il sistema avviene tramite il protocollo *HID* (*Human Interface Device*), simulando una tastiera. Quelli più complessi, invece, operano anche con *NFC* (*Near Field Communication*) o tramite *Bluetooth* e, di conseguenza, possono essere utilizzati anche con gli *smartphone*. Lo standard è già conforme ai livelli crittografici più elevati, come il *FIPS* (*Federal Information Processing Standard*) *140-2* [10] e il livello 3 di autenticazione (*AAL3*) indicato nelle linee guida *NIST SP800-63B*. Il *FIPS 140-2* è uno standard di sicurezza informatica del governo degli Stati Uniti utilizzato per convalidare i moduli crittografici. È conforme, inoltre, anche alla direttiva europea *PSD2* in merito alla *Strong Customer Authentication* (*SCA*).

Vantaggi:

- **Sicurezza.** Solo la persona che possiede il *token hardware* può connettersi all'account con successo. Ovviamente anche questi dispositivi sono soggetti a vari tipi di attacchi, per esempio la manomissione prima della consegna all'utente ma, a differenza degli altri metodi di autenticazione a due fattori, risultano molto più sicuri.
- **Usabilità.** È sufficiente collegarlo ad un sistema compatibile. Dopo una rapida configurazione, bisogna soltanto connettere il dispositivo e premere il bottone presente su quest'ultimo per procedere con l'autenticazione.

Svantaggi:

- **Supporto limitato.** Essendo una tecnologia recente, non è ancora utilizzabile con tutti i servizi.
- **Oggetto fisico.** È fondamentale ricordarsi di averlo sempre con sé, altrimenti non sarà possibile accedere all'account in questione. Bisogna sottolineare, però, che durante la configurazione viene richiesta automaticamente una procedura di backup ed è consigliabile possedere due *U2F security key* separate. La prima sarà quella che portiamo sempre con noi mentre la seconda servirà come chiave di backup da conservare in un luogo sicuro.
- **Costo.** Il prezzo oscilla dai 10\$ fino a 50–60\$ in base al modello. Non è certamente un'alternativa economica e, dal punto di vista delle aziende, la consegna di questi oggetti risulta abbastanza costosa.

2.3.2 SMS Text-Message & Voice-based

Nell'autenticazione a due fattori basata su *SMS* (*Short Message Service*), il sistema, dopo aver verificato il nome utente e la password inseriti, invia un codice utilizzabile una sola volta (*OTP*) tramite un *SMS* al numero di cellulare dell'utente. Per accedere, sarà necessario inserire il codice presente nell'*SMS*.

Nella modalità *voice-based*, invece, il sistema effettua una chiamata al telefono dell'utente e comunica verbalmente il codice da inserire per eseguire l'accesso. Nonostante sia un metodo poco diffuso, viene ancora utilizzato nei Paesi in cui il servizio telefonico non è ottimo o in cui il costo dei telefoni cellulari è molto elevato.

La modalità basata su *SMS* è senza dubbio la più diffusa ma è anche la meno sicura. Ciò a causa delle vulnerabilità del protocollo *SS7* (*Signaling System No. 7*): si tratta di un insieme standardizzato di protocolli di segnalazione usato nelle reti telefoniche *PSTN* (*Public Switched Telephone Network*) mondiali per gestire l'attivazione e la chiusura delle chiamate, i servizi *SMS* e molto altro. Questo standard presenta delle vulnerabilità, mai risolte, sfruttate anche per l'intercettazione delle comunicazioni. Ciò è decisamente preoccupante se si pensa che l'ultima revisione di *SS7* avvenne nel 1993 e che è tuttora in uso sia per i servizi di telefonia fissa che mobile (fino al 5G incluso). Per maggiori informazioni sulle vulnerabilità di *SS7*, consultare il report stilato dalla *FIGI* (*Financial Inclusion Global Initiative*) [9].

Oltre a possibili intercettazioni, questo metodo è vittima di una truffa molto frequente negli ultimi anni conosciuta come *SIM swapping*: in breve, si tratta di una tecnica di attacco che consente di trasferire da una *SIM card* ad un'altra un certo numero di telefono. Ciò permette a malintenzionati di avere accesso al numero di telefono del legittimo proprietario e, perciò, di ricevere gli *SMS* con i codici di autenticazione a due fattori corretti. Per compiere ciò, solitamente vengono utilizzate tecniche di *social engineering* per acquisire informazioni sulla vittima e per indurre gli operatori di telefonia mobile a emettere una nuova *SIM card*. Altre volte, invece, ci può essere la complicità dell'operatore o l'uso di un documento falso.

2.3.3 Token Software

Il funzionamento è pressoché identico ai *token hardware*. L'unica differenza è che il codice viene generato da un'applicazione installabile sui dispositivi mobili o sui dispositivi *desktop*.

Durante l'attivazione del 2FA su un sistema digitale, è necessario eseguire l'abbinamento dell'applicazione all'account dell'utente. In genere, ciò avviene tramite la scansione di un *QR code* che compare sullo schermo del computer durante la procedura di attivazione. In questo modo viene scambiata una chiave crittografica tra l'applicazione sul dispositivo e il sito in questione. Questa corrispondenza permetterà la generazione di *OTP* validi durante le successive autenticazioni da parte dell'utente.

Solitamente queste applicazioni restituiscono un codice a 6 cifre della durata di 30 secondi generato da un algoritmo che utilizza una chiave segreta e il *timestamp* corrente come input (*TOTP*, *Time-Based One-Time-Password*).

Risulta una valida alternativa alla modalità basata su *SMS* in quanto presenta numerosi vantaggi:

- **Supporto offline.** Né gli input degli algoritmi, né la generazione e la verifica dei codici richiedono una connessione ad Internet.

- **Senza PII.** Il numero di telefono dell'utente viene definito dal *NIST* come *Personally Identifiable Information* (*PII*). Con questa modalità di *2FA*, non viene richiesta la registrazione di nessun *PII*.
- **Metodi standardizzati.** Gli algoritmi utilizzati sono standard aperti definiti dall'*IETF* (*Internet Engineering Task Force*).
- **Basato sul software.** Non è dipendente dalle tariffe dell'operatore telefonico e dall'erogabilità dei messaggi.
- **Velocità.** Il tempo medio di autenticazione è minore (in base alla Figura 2 di un recente studio sull'usabilità dei metodi di autenticazione a due fattori [11]).
- **Sicurezza.** Non è richiesta una copertura telefonica e la chiave segreta viene condivisa una sola volta. La modalità basata su *SMS*, inoltre, può essere vittima di *SIM swapping* e non assicura che il messaggio sia letto solo dal legittimo proprietario.

Sul mercato sono presenti svariate applicazioni che permettono l'utilizzo di questo metodo di autenticazione. Le più note sono: *Google Authenticator*, *Microsoft Authenticator* e *Authy*. Questa funzionalità, tra l'altro, è disponibile anche in alcuni *password manager*.

2.3.4 Push notification

Questo tipo di autenticazione non prevede l'invio e l'inserimento di un *token*. Richiede, invece, che i sistemi digitali inviino una notifica *push* alla corrispondente applicazione presente tipicamente sul telefono dell'utente. In questo modo si può regolamentare l'accesso all'account.

Questa modalità funziona solo se il dispositivo è connesso a Internet ed è in grado di installare applicazioni. Elimina, inoltre, le vulnerabilità presenti nel metodo basato sugli *SMS*, creando una connessione diretta e sicura tra il servizio *2FA*, il dispositivo e il sistema in cui l'utente sta effettuando l'autenticazione.

2.3.5 Riconoscimento biometrico

In questo caso si utilizza l'utente come se fosse un vero e proprio *token umano*. È un metodo molto recente ma, fra qualche anno, sarà sicuramente quello più utilizzato. Permette la verifica dell'identità di una persona tramite riconoscimento facciale, impronte digitali e lettura della retina. Sono ancora in via di sviluppo, invece, approcci di verifica basati sulla voce, sulla calligrafia e sul modo con cui un utente digita sulla tastiera.

2.4 Algoritmi utilizzati nel 2FA

Attualmente, vengono utilizzati principalmente due tipi di algoritmi per implementare l'autenticazione a due fattori: *HOTP* (*HMAC-based One-Time Password*) oppure *TOTP* (*Time-based One-Time Password*).

Il primo costituisce lo standard originale su cui, successivamente, venne basato *TOTP*. Entrambi i metodi utilizzano una chiave segreta come primo input ma, mentre *TOTP* usa la data e l'ora attuale come secondo input, *HOTP* si avvale di un contatore che viene incrementato ad ogni nuova validazione.

Forniscono entrambi un metodo di autenticazione che permette la generazione simmetrica di codici che posso essere utilizzati soltanto una volta (*OTP, one-time password*). Queste password dinamiche sono difficili da violare ed aggirare in quanto l'attaccante avrà bisogno di accedere sia a qualcosa che l'utente possiede (un dispositivo che genera *OTP*, come un *token hardware* o uno *smartphone*) sia a qualcosa che l'utente conosce (come un codice *PIN*).

Non necessitano di una connessione ad Internet per il corretto funzionamento ed è sufficiente che l'utente inserisca nella piattaforma web il codice generato dall'applicazione presente sul suo dispositivo. Il sistema digitale, a sua volta, calcola un codice usando lo stesso algoritmo: se i codici corrispondono, allora all'utente verrà consentito l'accesso.

2.4.1 HMAC

Per comprendere al meglio *HOTP* e *TOTP* è necessario fare un passo indietro. Dato che entrambi gli algoritmi utilizzano *HMAC* (*Hash-based Message Authentication Code*), è bene comprendere velocemente il suo funzionamento.

HMAC utilizza due funzioni separate: **hash** e **MAC**. L'**hash** è una funzione matematica che produce una sequenza di bit, detta *digest*, strettamente correlata ai dati di ingresso. Si tratta di una funzione non invertibile che mappa una stringa di lunghezza arbitraria in una stringa di lunghezza predefinita.

L'*hash* è deterministica, cioè l'algoritmo produrrà sempre lo stesso output a partire da uno stesso input. Deve essere impossibile, inoltre, ottenere la stringa originale a partire dal suo *digest*. Vi è un vantaggio anche dal punto di vista computazionale in quanto confrontare o processare i *digest* è molto più facile che confrontare i file originali. Grazie a queste sue proprietà, l'*hashing* viene utilizzato per la compressione di file, l'indicizzazione di dati, la generazione di *checksum*, ecc.

MAC (*Message Authentication Code*), invece, è un blocco di dati (*checksum* crittografico) utilizzato per garantire l'autenticazione e l'integrità di un messaggio digitale. L'algoritmo riceve in input una chiave segreta e un messaggio da autenticare di lunghezza arbitraria e restituisce in output un *MAC*. Precondizione per il corretto funzionamento è la condivisione tra mittente e destinatario di una chiave privata comune e l'utilizzo dello stesso algoritmo di generazione del *MAC*.

Il mittente invierà il messaggio in chiaro e il *MAC* calcolato in base alla sua chiave segreta. Il destinatario calcolerà il *MAC* a partire dal messaggio ricevuto e dalla sua chiave. Se il *MAC* inviato dal mittente e quello appena calcolato dal destinatario coincidono, allora il messaggio risulta autenticato e integro. L'algoritmo, quindi, permette di verificare l'integrità dei dati, l'autenticità del mittente del messaggio ma non la confidenzialità delle informazioni contenute. I *MAC*, inoltre, non offrono la proprietà del non ripudio, in quanto ogni utente che può verificare un *MAC* è anche capace di generare *MAC* per altri messaggi.

HMAC, dunque, combina queste due funzioni. In base all'algoritmo scelto, viene fatto l'*hash* del messaggio in blocchi di dimensione B e il *digest* prodotto avrà dimensione L. Questo significa che *HMAC* utilizza solo algoritmi di *hashing* che cifrano i dati a blocchi (*block ciphers*). Viene generata, successivamente, una chiave segreta in modo casuale e quest'ultima dev'essere condivisa con il destinatario tramite un canale sicuro. Se la chiave risulta più lunga della dimensione B del blocco, ne verrà fatto l'*hashing* per ottenere una chiave di dimensione L. Se, invece, la chiave è più corta della dimensione

B del blocco, allora verranno aggiunti una serie di zero a destra (*padding*) affinché la nuova lunghezza della chiave sia B.

Dopodiché è necessario generare altre due chiavi: la chiave interna (*inner key*) e la chiave esterna (*outer key*). La prima viene generata eseguendo lo *XOR* bit a bit (*bitwise XOR*) tra la chiave segreta e l'*ipad* (0x36 ripetuto B volte). La chiave esterna, allo stesso modo, viene generata eseguendo un *bitwise XOR* tra la chiave segreta e l'*opad* (0x5c ripetuto B volte). In seguito, viene accodato il messaggio alla chiave interna e il tutto viene processato dall'algoritmo di *hashing* scelto che produrrà un *digest* di lunghezza L. Quest'ultimo valore verrà a sua volta accodato alla chiave esterna e su ciò che ne risulta verrà nuovamente effettuato l'*hashing*.

Il risultato finale, riassumendo, viene ottenuto nel seguente modo:

$$\text{HMAC}(K, m) = H\left((K \oplus \text{opad}) \parallel H((K \oplus \text{ipad}) \parallel m)\right)$$

Nelle precedente formula, H è la funzione di *hashing*, m è il messaggio da autenticare, K è la chiave segreta, \oplus rappresenta il *bitwise XOR* e \parallel identifica l'operazione di concatenazione.

La stringa ottenuta verrà usata per verificare l'integrità e l'autenticità del messaggio. Il destinatario utilizzando il messaggio ricevuto e la chiave segreta che possiede andrà ad eseguire lo stesso algoritmo del mittente e confronterà l'*hash* ottenuta con l'*hash* che ha ricevuto. In questo modo, se le due stringhe combaciano significa che il messaggio non è stato manipolato e che, essendo la chiave un segreto che solo le due parti conoscono, l'identità del mittente è stata verificata.

2.4.2 HOTP

Si tratta di uno standard aperto documentato nel *RFC 4226* da parte dell'*IETF* che utilizza l'algoritmo *HMAC* (*Hash-based Message Authentication Code*) per la generazione di *OTP*.

L'autenticazione tramite *HOTP* richiede che entrambe le parti (il sistema e il dispositivo) abbiano stabilito alcuni parametri:

- una funzione hash, H
- una chiave segreta, K
- un contatore, C
- la lunghezza del codice restituito in output (il valore di default è 6).

La chiave segreta dev'essere una stringa pseudocasuale codificata in *Base32* e, a seguito delle indicazioni contenute in *RFC 4226*, deve avere una lunghezza di almeno 128 bit. Le chiavi, inoltre, devono essere protette dall'uso non autorizzato.

Per quanto riguarda il contatore, il dispositivo incrementa la sua variabile contatore ogni volta che l'utente genera un nuovo codice. Il sistema, però, non può sapere quante volte l'utente decide di generare il codice, perciò incrementa il suo contatore solo quando un utente inserisce un codice di autenticazione corretto. Di conseguenza, il contatore del dispositivo e quello del sistema non sono, in genere, sincronizzati. È possibile, pertanto, che i codici generati non risultino uguali.

Per ovviare a questo problema, il sistema dovrà incrementare il suo contatore fino a quando il codice generato risulterà uguale a quello del dispositivo. Tuttavia, se dopo un certo numero di incrementi (parametro di *look-ahead*) non viene trovato il codice corretto, l'autenticazione fallisce e il contatore viene portato al suo valore iniziale. Il numero di incrementi possibili deve essere necessariamente limitato in quanto, se l'utente inserisce un codice sbagliato, il sistema continuerà ad incrementare il valore indefinitamente.

Entrambe le parti calcolano il codice di autenticazione generando un *HMAC* a partire dal contatore e dalla chiave segreta.

$$HOTP(K, C) = HMAC(K, C)$$

La stringa che si ottiene, però, risulta troppo lunga: anche utilizzando una funzione di hashing di tipo *SHA 1*, non più sicura al giorno d'oggi, si ottengono 160 bit (20 byte). È necessario ridurla in modo tale che consista di almeno 6 cifre così da rendere il codice facilmente inseribile dall'utente ma richiedendo, allo stesso tempo, una certa robustezza contro possibili attacchi esterni atti a scoprire l'*OTP*.

Il processo di troncamento è abbastanza artificioso. Partendo dall'hash in formato esadecimale, questo viene diviso in gruppi di dimensione di 1 byte. In questo modo, dato che una cifra esadecimale corrisponde a 4 bit, i gruppi saranno composti da 2 cifre esadecimali e ognuno sarà contraddistinto da un indice.

Il prossimo passaggio consiste nel considerare l'ultima cifra esadecimale come un valore di *offset*. Vengono estratte le cifre esadecimali a partire dall'indice che ha come valore l'*offset* precedentemente calcolato e fino ai successivi tre indici. Il numero estratto è chiamato *Dynamic Binary Code (DBC)*.

Dopodiché, viene eseguito l'*AND* bit a bit tra questo numero e 0x7f, che essendo in binario 01111111, permettere di *mascherare* (portare a 0) il primo bit del *DBC*. Ciò è necessario perché, quando viene svolta l'operazione modulo, processori diversi effettuano il modulo con segno e il modulo senza segno in modi diversi. Il primo bit identifica il segno del numero e il suo annullamento permette di risolvere quest'inconveniente.

Il risultato di quest'ultima operazione viene poi convertito in un numero decimale. Infine, viene calcolato il modulo tra il numero ottenuto al passo precedente e 10^d (d rappresenta il numero di cifre che si vuole ottenere nel codice finale). Se la stringa ottenuta non risulta essere di sei cifre, allora viene aggiunto uno zero in testa.

2.4.3 TOTP

TOTP (Time-based One-time Password) è un algoritmo che venne presentato nel 2008 da *OATH (Initiative for Open Authentication)* come un'espansione di *HOTP* e che nel 2011 divenne uno standard aperto documentato nel *RFC 6238*. Permette la generazione di *OTP* a partire da una chiave segreta condivisa K e dal timestamp T corrente usando una funzione di hash H .

Per effettuare l'autenticazione, entrambe le parti devono stabilire i parametri usati in *HOTP* e:

- T_0 , lo *Unix time* a partire dal quale iniziare a contare gli intervalli temporali (di default è 0),
- T_X , un intervallo che specifica per quanto tempo i codici generati saranno validi (di default vale 30 secondi)

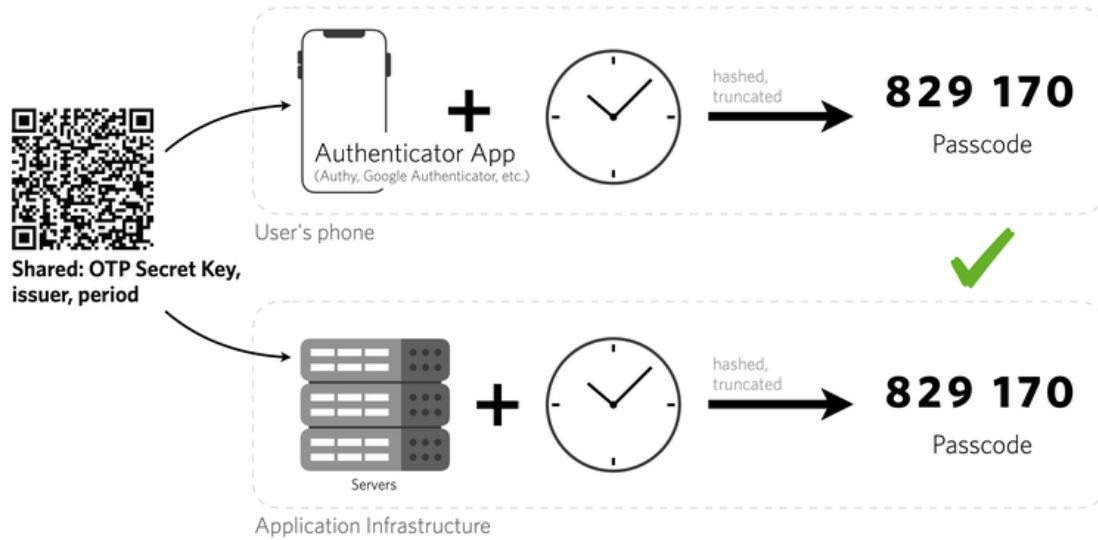


Figura 2.2: Processo di autenticazione a due fattori basato su TOTP (da [12])

TOTP utilizza l'algoritmo *HOTP* sostituendo il contatore con un valore C_T basato sulla data e l'ora attuale.

$$C_T = \left\lfloor \frac{T - T_0}{T_X} \right\rfloor$$

$$\text{TOTP}(K) = \text{HOTP}(K, C_T)$$

T è il timestamp corrente in secondi a partire da una certa epoca, T_0 è l'epoca specificata in secondi rispetto all'epoca *Unix*¹ (utilizzando lo *Unix time*, T_0 vale 0) mentre T_X indica per quanti secondi un codice dovrà rimanere valido. C_T , dunque, è il numero di intervalli di T_X secondi compresi tra T_0 e T . Per esempio, se $T_X = 30$ e $T_0 = 0$ allora:

- $C_T = 0$ se $0 \leq T \leq 29$
- $C_T = 1$ se $30 \leq T \leq 59$
- $C_T = 2$ se $60 \leq T \leq 89$
- ...

In genere un valore di T_X elevato si traduce in un intervallo temporale maggiore in cui un codice può essere accettato ma ciò rende l'algoritmo più vulnerabile ad attacchi esterni. Il valore di ogni parametro, perciò, dovrebbe essere un compromesso tra usabilità e sicurezza.

Il processo di validazione è identico a quello di *HOTP* ma, l'uso del timestamp corrente come parametro, fa sorgere alcuni problemi. Se il client invia un codice che è in procinto di scadere, il server, a causa della latenza della rete, potrebbe processare la richiesta nell'intervallo successivo, generando un codice diverso e facendo fallire l'autenticazione. Ciò rende necessaria l'introduzione di una *delay window* (s) che faccia in modo che il server non controlli solo il codice generato tramite l'intervallo corrente ma

¹Nei sistemi operativi basati su *Unix* il termine epoca fa riferimento alla mezzanotte (*UTC*) del 1° gennaio 1970

anche tutti i codici calcolati a partire dagli s intervalli precedenti e successivi. Di norma $s = 1$ risulta un buon compromesso per non rendere l'algoritmo troppo esposto ad attacchi esterni.

Un secondo problema riguarda la sincronizzazione. Gli orologi interni del client e del server tendono, molto lentamente, a desincronizzarsi fino al punto che neanche la *delay window* può consentire una corretta validazione. È necessario un meccanismo di resincronizzazione. Ad esempio, il client potrebbe inviare più codici consecutivi e il server dovrebbe validare la sequenza ricevuta entro un certo numero di intervalli temporali. Ciò è necessario in quanto per un attaccante risulta molto più complicato indovinare una sequenza di codici validi rispetto ad un singolo codice. Deve essere precisato, tuttavia, un valore massimo di intervalli temporali esaminabili dal server per evitare attacchi di tipo *DoS* (*denial-of-service*). Non appena il server trova una corrispondenza per la sequenza di codici, può essere determinato di quanto i due orologi sono sfasati in modo che ne venga tenuto conto nella validazioni di nuovi *OTP*.

Un'altra soluzione che a differenza della precedente non richiede una resincronizzazione manuale prevede che il server calcoli lo sfasamento con il client ad ogni validazione avvenuta con successo.

Bisogna, però, anche tenere in considerazione come difendersi contro attacchi di tipo *brute-force*. Per proteggere il sistema da un attaccante che può effettuare molti tentativi di autenticazione da diversi dispositivi si può procedere in almeno due modi. Il primo prevede di impostare un parametro che specifichi un numero massimo di tentativi di autenticazione possibili prima che l'account non sia più accessibile. Il secondo, invece, stabilisce un ritardo con *backoff esponenziale* tra più tentativi di validazione consecutivi. Pertanto, la probabilità che un attaccante riesca ad indovinare il codice corretto può essere calcolata nel seguente modo:

$$P = \frac{s \times v}{10^d}$$

Nella formula precedente s è la *delay window*, v è il numero di tentativi permessi e d è la lunghezza dei codici generati. Da questa espressione si può osservare il ruolo che i parametri ricoprono nella robustezza dell'algoritmo. Per diminuire la probabilità di successo in caso di un attacco esterno è necessario aumentare la lunghezza dei codici generati e/o espandere l'alfabeto usato per creare gli *OTP*.

TOTP, dunque, risulta una valida alternativa rispetto a *HOTP* per una serie di ragioni. Il problema principale di *HOTP* è che i codici generati non hanno una scadenza: rimangono validi fino al loro utilizzo o fino a quando il server non ne convalida uno nuovo. Ciò rende il metodo di autenticazione poco sicuro in quanto vulnerabile ad attacchi (per esempio di tipo *brute-force*) da parte di malintenzionati.

HOTP, inoltre, presenta il problema della desincronizzazione del contatore presente nel server. Come già detto precedentemente, il problema è parzialmente risolvibile con il parametro di *look-ahead*. Tuttavia se qualcuno, intenzionalmente (*hacker*) o no, generasse un numero molto elevato di codici sul dispositivo, il *token* di autenticazione diventerebbe inutile. *TOTP*, al contrario, genera codici che rimangono validi soltanto per un certo numero di secondi (di solito 30). Perciò, anche se venissero scoperti da un attaccante, quest'ultimo non riuscirebbe ad accedere all'account (a meno che il malintenzionato non riesca a scoprire e ad inserire i codici prima dell'utente stesso). In aggiunta, non è presente il problema della sincronizzazione delle variabili tra il dispositivo e il server. L'unico svantaggio è lo sfasamento dei *clock* ma sono state proposte molte soluzioni per risolvere questo problema.

3

Analisi del problema

L'obiettivo dell'attività di tirocinio consiste nella progettazione e nell'implementazione di un servizio di autenticazione a due fattori all'interno di una piattaforma web. Ciò consentirà un ulteriore rafforzamento della sicurezza degli account degli utenti. Oggigiorno, infatti, non è più sufficiente impostare password forti ed univoche: un'autenticazione che si basa solo su password non è sicura in quanto la sicurezza dell'*account* dipende da un solo fattore. Lo scopo, dunque, è progettare una modalità di autenticazione basata sull'uso di due fattori.

3.1 Requisiti

La raccolta e l'analisi dei requisiti risulta una delle fasi più importanti dello sviluppo di un *sistema software*. Il risultato del progetto, infatti, dipende fortemente dai requisiti raccolti in questa fase.

Non è sempre necessario procedere fin da subito alla stesura di una dettagliata specifica dei requisiti che descrive le funzionalità del nuovo prodotto nella loro interezza: dipende dall'approccio di sviluppo scelto e dal sistema software da progettare. Inizialmente, però, risulta sempre utile stilare una breve descrizione dei servizi che il sistema dovrà fornire e dei vincoli operativi da rispettare nelle fasi successive.

I *requisiti software* possono essere classificati in due macro-aree: **requisiti funzionali** e **requisiti non funzionali**. Nei prossimi paragrafi verranno descritti i principali requisiti del progetto in base alla precedente distinzione cercando, per quanto possibile, di evitare ambiguità e inconsistenze.

3.1.1 Requisiti funzionali

Questa tipologia di requisiti permette di descrivere le funzionalità che il nuovo prodotto deve offrire. Delineano, inoltre, come il sistema deve reagire a specifici tipi di input e in situazioni particolari.

1. Il servizio deve permettere agli utenti registrati al sistema di attivare un secondo fattore di autenticazione da utilizzare durante la procedura di *login*.
2. Il servizio deve poter essere attivabile in maniera opzionale.
3. Il servizio deve poter essere disattivato in qualunque momento.

4. L'utente deve poter scegliere tra più modalità di autenticazione. Ciò consente di avere almeno una procedura di *backup* in caso di problemi o imprevisti.
5. Il servizio deve permettere all'utente di confermare esplicitamente l'attivazione dell'autenticazione a due fattori. Il processo di attivazione, dunque, dovrà consistere di almeno due fasi: la prima di inizializzazione e la seconda che si occuperà di attivare effettivamente il servizio.

3.1.2 Requisiti non funzionali

Si tratta di requisiti che pongono una serie di vincoli su come il sistema debba operare (non sono correlati, dunque, all'aspetto funzionale del *software*). Permettono di descrivere le proprietà che il prodotto deve avere in relazione a determinati servizi o funzioni. A titolo di esempio, questo tipo di requisiti consente di esprimere le caratteristiche del processo di sviluppo e il livello di efficienza, affidabilità e robustezza richiesti al sistema.

1. Il servizio deve essere sviluppato usando *PHP* come linguaggio di programmazione e *Symfony* (versione 4.4) come *framework* principale.
2. Il sistema deve utilizzare una tipologia di *2FA* che sia il miglior compromesso tra sicurezza e usabilità. È necessaria, quindi, un'analisi approfondita delle attuali modalità di autenticazione.
3. Il servizio deve essere modulare, ovvero composto da un certo numero di sottosistemi ognuno dei quali il più possibile indipendente dagli altri e che svolge un compito ben definito. Ciò aumenta la comprensibilità del sistema e rende più facili la modifica ed il riuso di quest'ultimo.
4. Il servizio deve essere in grado di interfacciarsi con altri sistemi *software*.
5. Il servizio deve avere un basso grado di accoppiamento con i moduli presenti nel sistema in cui verrà aggiunto. Questo requisito è indispensabile in quanto, come accennato nel requisito precedente, il sistema deve poter essere aggiunto facilmente anche su altre piattaforme web.
6. I tempi di risposta del sistema devono essere sufficientemente contenuti sia durante l'attivazione del *2FA* che durante l'autenticazione (si richiedono tempi inferiori ai 10 secondi).
7. Il servizio deve essere ragionevolmente robusto, ovvero in situazioni impreviste (inserimento di input scorretti, alterazione del flusso di autenticazione o fallimenti dei componenti software esterni ad esso) il comportamento del sistema deve essere *ragionevole*.
8. Il servizio deve essere utilizzabile anche da applicazioni sviluppate per dispositivi *mobile*.
9. Deve essere sviluppata un'interfaccia utente veloce ed intuitiva da utilizzare. Gli utenti dovranno essere capaci di attivare ed usare il servizio di autenticazione in autonomia e senza incorrere in procedure complesse.

L'ultimo requisito merita un ulteriore approfondimento in quanto la sicurezza verrà rafforzata solo se il servizio sarà attivato dagli utenti del sistema. Non è sufficiente, pertanto, offrire un servizio di autenticazione a due fattori: bisogna fare in modo che venga anche adottato. A tal proposito, secondo

uno studio effettuato nel 2019 sull'usabilità dei metodi di autenticazione a due fattori [11], solo il 29% delle persone ritiene che il tempo e la complessità aggiunti dalle procedure di *2FA* nel processo di *login* valgano veramente la sicurezza aggiuntiva. Spesso gli utenti prediligono la velocità e la praticità di utilizzo rispetto alla sicurezza.

Negli ultimi anni molte aziende hanno dovuto affrontare il problema di come convincere gli utenti ad attivare il *2FA*. Una delle soluzioni più diffuse consiste nell'offrire un qualche vantaggio all'utente che attiva l'autenticazione multifattoriale. Si tratta di un metodo proposto inizialmente dalle aziende videoludiche ma che attualmente viene utilizzato anche in altri settori. Altre soluzioni prevedono di inserire dei promemoria nelle schermate di *login* e di fornire all'utente più modalità di autenticazione tra cui scegliere. Rendere, invece, obbligatorio l'utilizzo del *2FA* è una soluzione praticabile solo per certi tipi di servizi (di solito finanziari) ritenuti particolarmente importanti dagli utenti. L'adozione di questa tecnica, infatti, potrebbe far perdere una buona percentuale di utenza del sistema.

Tornando al progetto, le soluzioni sopra citate saranno prese in considerazione una volta terminata l'implementazione base del servizio e dopo aver analizzato come l'utenza abbia reagito all'introduzione del *2FA*. Nonostante ciò, è sicuramente vantaggioso progettare fin da subito un'interfaccia utente intuitiva e veloce da utilizzare.

Progettazione della soluzione

Nel presente capitolo verrà discussa la fase di progettazione riguardante il servizio di autenticazione a due fattori da sviluppare. L'obiettivo consiste nel trasformare i requisiti identificati nel capitolo precedente (Capitolo 3) in uno schema architetturale che sia attuabile e sufficientemente robusto per il servizio da realizzare. Verranno discusse, quindi, tutte le scelte riguardanti l'architettura del sistema, la struttura dei componenti che lo costituiscono, le relazioni che avvengono tra le sue componenti interne e le connessioni con i sistemi esterni. Tutto ciò che verrà stabilito in questo stadio, infine, sarà usato come riferimento nella fase implementativa.

4.1 Selezione della modalità di 2FA

È necessario, innanzitutto, decidere quale modalità di *2FA* adottare. A tal proposito, il requisito non funzionale numero 2 afferma che la tipologia di *2FA* scelta deve essere il miglior compromesso tra sicurezza e usabilità. Nel Capitolo 2 sono state analizzate tutte le diverse modalità di autenticazione a due fattori e i corrispettivi vantaggi e svantaggi. Escludendo il riconoscimento biometrico che al momento è un'alternativa poco praticabile, le opzioni disponibili sono:

- **Messaggi SMS**
- **Token Hardware**
- **Token Software**
- **Notifiche push**

La modalità basata sui **messaggi SMS** ha il vantaggio di essere quella più utilizzata grazie alla sua facilità di utilizzo e familiarità. Al giorno d'oggi la maggior parte delle persone possiede un cellulare e una linea telefonica che permette l'invio e la ricezione di messaggi *SMS*. Non è necessario, quindi, né installare un'applicazione né dotarsi di una connessione ad internet. Oltretutto, secondo un report [3] stilato da *Okta* (un'azienda che offre servizi di autenticazione e autorizzazione basati sul *cloud*), l'uso di questa tipologia di *2FA* è cresciuto del 9% negli ultimi due anni.

Uno degli svantaggi più significativi, però, consiste nella possibilità di intercettare i messaggi *SMS* a causa di alcune vulnerabilità presenti nel protocollo *SS7*. Questa modalità, peraltro, oltre ad essere

vittima dello *SIM swapping*, richiede anche di condividere un'informazione personale (il numero di telefono) ai sistemi digitali in questione.

I **token hardware** risultano uno dei metodi di autenticazione più sicuri poiché, dato che si tratta di dispositivi fisici, fintanto che vengono tenuti al sicuro soltanto il proprietario potrà accedere all'account. Si tratta, oltretutto, di dispositivi facilmente usabili ma, essendo una tecnologia recente, non sono ancora supportati da tutti i servizi e il costo è abbastanza elevato. Inoltre, essendo dispositivi di piccole dimensioni, possono essere persi o dimenticati con facilità.

I **token software** non richiedono l'attivazione di una linea telefonica e, dopo l'installazione di un'applicazione di generazione di codici temporanei, non necessitano neanche di una connessione ad internet. Non è necessario, quindi, condividere il proprio numero di telefono e gli algoritmi utilizzati sono standard aperti definiti dall'*IETF* (*Internet Engineering Task Force*).

Le **notifiche push**, a differenza degli altri metodi, permettono semplicemente l'approvazione o il rifiuto della richiesta di accesso all'account. Non è necessaria la creazione di codici e consente di visualizzare anche l'indirizzo *IP* (*Internet Protocol Address*), il tipo del dispositivo e la posizione geografica approssimata da cui è stato eseguito il tentativo di login. Necessita, però, di una connessione ad internet e di un dispositivo in grado di installare applicazioni.

In base alle precedenti considerazioni, i *token software* e le *notifiche push* costituiscono entrambi un buon compromesso tra sicurezza e usabilità. I *token software*, tuttavia, risultano il metodo che ha ricevuto il miglior punteggio dai partecipanti ad uno studio sull'usabilità dei metodi di autenticazione a due fattori [11] e, in aggiunta, non necessitano di una connessione ad internet. Si è deciso, dunque, di adottare questa modalità di *2FA*.

Per quanto riguarda l'algoritmo da utilizzare per la generazione dei codici temporanei, **TOTP** risulta una valida alternativa rispetto a **HOTP** (come evidenziato nella sezione 2.4.3 del capitolo precedente) in quanto i codici generati hanno un breve periodo di validità e non è presente il problema della sincronizzazione dei contatori tra il server e il dispositivo.

Riassumendo, la tipologia di *2FA* adottata sarà quella basata su *token software* utilizzando *TOTP* come algoritmo di generazione dei codici temporanei.

4.2 Scelta del bundle 2FA

4.2.1 Perché utilizzare una libreria?

Prima di procedere con la selezione del *bundle 2FA* più opportuno, è doveroso spiegare perché si è optato per l'utilizzo di un *bundle* al posto di progettare il servizio da zero.

Il motivo principale è il tempo a disposizione per finire l'intero progetto: partire da zero e progettare ogni singola funzionalità avrebbe richiesto più di un mese di lavoro. In generale, inoltre, è poco sensato sviluppare dei servizi e delle funzionalità che sono già state progettate ed implementate. Le librerie, inoltre, hanno il vantaggio di essere già state testate consentendo, quindi, di incorrere in meno *bug* e problemi.

È importante, tuttavia, capire come funziona e cosa offre una libreria prima del suo utilizzo. Ciò consentirà di correggere eventuali *bug* ed errori che potrebbero sorgere durante la fase di integrazione e di adattare le funzionalità della libreria ai propri bisogni (è improbabile che una libreria fornisca una

soluzione adatta ad ogni situazione e sistema). Si tratta, infine, di una valida opportunità per imparare nuove strategie risolutive e architetturali.

Naturalmente l'adozione di una libreria presenta anche degli svantaggi. Alcune librerie, ad esempio, per poter funzionare correttamente hanno bisogno di altre librerie che a loro volta utilizzano altre librerie ancora, producendo un notevole *overhead*. Non si può essere certi, peraltro, che le librerie assicurino sempre la retro-compatibilità e che non contengano *bug*.

È ragionevole, quindi, utilizzare una libreria solo se si conosce in maniera approfondita il suo funzionamento. Detto questo, però, a parte sistemi specifici in cui le *performance* e la sicurezza sono un requisito imprescindibile, è sempre vantaggioso far affidamento ad una libreria (almeno in una prima fase progettuale).

4.2.2 Analisi dei bundle 2FA presenti sul mercato

Nella sezione precedente è stata selezionata la tipologia di *2FA* basata su *token software* utilizzando *TOTP* come algoritmo di generazione dei codici temporanei. In base al primo requisito non funzionale, inoltre, il progetto dovrà essere realizzato usando *PHP* come linguaggio di programmazione e *Symfony* (4.4) come *framework*. Il bundle da scegliere, dunque, dovrà fornire tutte queste funzionalità ed aderire ai requisiti precedentemente delineati.

In seguito ad una prima fase di ricerca, sono state selezionate tre possibili librerie:

- **sonata-project/GoogleAuthenticator**
- **scheb/2fa**
- **Verify API**

Verify API, ideata da *Twilio*, non è propriamente una libreria ma un' *API* (*Application Programming Interface*). Quest'ultima offre numerose modalità di autenticazione, inclusi i *token software* che utilizzano l'algoritmo *TOTP*. Si tratta di un servizio ben documentato che può gestire ingenti carichi di richieste (elevata scalabilità) e che offre svariate soluzioni per incrementare la sicurezza e per ovviare ad alcuni problemi dell'algoritmo *TOTP* (come la desincronizzazione tra il *clock* del dispositivo e quello del server). Lo svantaggio principale, tuttavia, consiste nel fatto che non si può conoscere come sono stati progettati ed implementati i servizi e le funzionalità offerte. Non si tratta, dunque, di un progetto *open-source* e costringe il servizio in via di sviluppo a dipendere fortemente da enti di terze parti (*Twilio*). *Verify API* consente, dunque, di gestire l'autenticazione a due fattori in tutti i suoi aspetti e il costo del servizio viene calcolato in base al numero di autenticazioni avvenute con successo. Si tratta, dunque, di una soluzione adatta a sistemi di grandi dimensioni che richiedono un servizio completo e pronto all'uso.

Il bundle **scheb/2fa**, progettato da Christian Scheb, fornisce un servizio di autenticazione a due fattori compatibile con applicazioni *Symfony*. È suddiviso in pacchetti e in tal modo permette di installare soltanto le funzionalità strettamente necessarie al progetto. Consente, inoltre, una notevole flessibilità nell'adattare i servizi del *bundle* ad un'applicazione specifica e fornisce molte funzionalità aggiuntive tra cui:

- Codici *QR* per trasferire, in fase di attivazione, il codice segreto al dispositivo *mobile*

- Codici di backup monouso in caso di imprevisti con il dispositivo *mobile*
- Condizioni personalizzabili che definiscono quando eseguire l'autenticazione a due fattori
- Protezione contro attacchi *CSRF* (*Cross-Site Request Forgery*) che sfruttano la fiducia di un sito nel browser di un utente.

In aggiunta, fornisce sia un pacchetto per implementare il *2FA* in modo che sia utilizzabile con l'applicazione *Google Authenticator* (codice a 6 cifre con un periodo di validità di 30 secondi e *SHA1* come algoritmo di *hashing*) sia un pacchetto che permette di modificare molti dei parametri usati dall'algoritmo *TOTP*. La libreria possiede una buona documentazione e, a partire dalla versione compatibile con *Symfony* 4.4, le versioni successive sono supportate ed aggiornate attivamente.

La libreria **sonata-project/GoogleAuthenticator** fa parte di un progetto *open-source* che permette di integrare *Google Authenticator* in un'applicazione *PHP*. Consente di validare i codici temporanei inseriti dagli utenti (quindi di consentire o rifiutare l'accesso al sistema) e di generare il codice *QR* che l'utente utilizzerà per condividere il codice segreto con *Google Authenticator*. La libreria, però, presenta dei problemi di progettazione e alcuni *bug* critici. Dato che, al momento, non è presente alcun tipo di supporto per la risoluzione di questi problemi, il *bundle* è stato dichiarato abbandonato. Di conseguenza non è consigliabile l'utilizzo di questa libreria.

Dopo aver analizzato singolarmente i vantaggi e gli svantaggi delle precedenti librerie, si è deciso di optare per l'utilizzo del *bundle* **schneb/2fa**. Quest'ultimo, infatti, presenta i seguenti vantaggi:

- progetto *open-source*
- supporto attivo
- utilizzabile senza costi aggiuntivi
- compatibile con il *framework* *Symfony* 4.4
- offre un metodo alternativo di autenticazione (codici di backup) in caso di imprevisti (requisito funzionale numero 4)

4.3 Analisi delle funzionalità del servizio

Il servizio dovrà permettere tre attività principali: l'attivazione del *2FA*, la disattivazione del *2FA* e l'autenticazione tramite *2FA*. L'analisi di queste funzioni consentirà, inoltre, di individuare i singoli componenti che costituiranno il servizio.

Tenendo in considerazione la tipologia di *2FA* selezionata e il *bundle* scelto nella precedente sezione, per procedere all'attivazione sarà necessario collegare il proprio account all'applicazione generatrice di codici temporanei presente sul proprio dispositivo mobile (*authenticator*). Per questo motivo è richiesta la creazione e la condivisione di un codice segreto tra il server e il dispositivo. Per condividere il codice, si possono seguire due approcci:

1. l'utente digita manualmente il codice nell'*authenticator*

2. il codice viene trasferito in automatico tramite la scansione di un codice *QR* che contiene tutte le informazioni necessarie

Dato che praticamente tutti i moderni *authenticator* permettono la scansione di un codice *QR*, per rendere il servizio più veloce e usabile è opportuno scegliere la seconda alternativa. Fatto ciò, l'applicazione genererà in automatico codici temporanei di autenticazione per permettere all'utente di eseguire l'accesso al suo account.

Il quarto requisito funzionale, però, dichiara la necessità di fornire più modalità di autenticazione con cui eseguire l'accesso. Dal momento che il *bundle* scelto lo consente, si potrebbero fornire all'utente dei codici di backup monouso che consentano l'accesso al sistema anche senza inserire il codice temporaneo fornito dall'apposita applicazione. Il servizio dovrà generare e fornire all'utente 10 codici di backup durante il processo di attivazione del *2FA*. È necessario sottolineare, tuttavia, che si tratta di una procedura da utilizzare in caso di problemi o imprevisti e non dovrebbe essere intesa come la principale modalità di autenticazione.

Il quinto requisito funzionale, inoltre, sottolinea il fatto che il processo di attivazione del *2FA* deve avvenire in due fasi distinte per permettere all'utente di confermare la propria scelta ed attivare effettivamente il servizio. Si potrebbe, dunque, consentire l'inserimento di un codice temporaneo in fase di attivazione che permetta di rendere operativo il servizio di autenticazione. L'inserimento di questo codice determina, quindi, la conclusione della fase di inizializzazione e l'inizio della fase di attivazione vera e propria.

Una volta conclusa con successo la fase di attivazione, l'utente in fase di login dovrà prima autenticarsi normalmente tramite nome utente e password. Solo dopo che il sistema esterno ha attestato la validità delle credenziali, il servizio *2FA* controllerà se l'utente ha attivato l'autenticazione a due fattori. In caso positivo, l'accesso e i privilegi verranno temporaneamente negati e verrà richiesto all'utente un codice di autenticazione temporaneo generato dall'applicazione presente sul suo dispositivo (l'utente è parzialmente autenticato). Se il codice è corretto, l'utente potrà accedere al suo account.

Un utente, infine, potrà disattivare il servizio di autenticazione a due fattori soltanto dopo essersi autenticato con successo. Il servizio procederà all'invalidazione di tutti i codici di backup assegnati all'utente e del codice segreto condiviso con il dispositivo. Al successivo login, dunque, l'utente potrà accedere al suo account fornendo soltanto nome utente e password.

4.4 Diagramma dei casi d'uso

Il diagramma 4.1 consente di illustrare come le entità esterne (attori) interagiranno con il sistema e permette di visualizzare e organizzare le funzioni (casi d'uso) che il sistema dovrà mettere a disposizione per raggiungere gli obiettivi delineati nei requisiti funzionali e nella precedente sezione. È importante sottolineare, però, che si tratta di un diagramma che fornisce soltanto una visione ad alto livello del sistema.

Sulla sinistra sono raffigurati gli attori principali (utente registrato e utente in fase di login) che interagiscono direttamente con il sistema, mentre sulla destra vengono rappresentati gli attori secondari (*database* della piattaforma web esterna) che interagiscono indirettamente con il sistema.

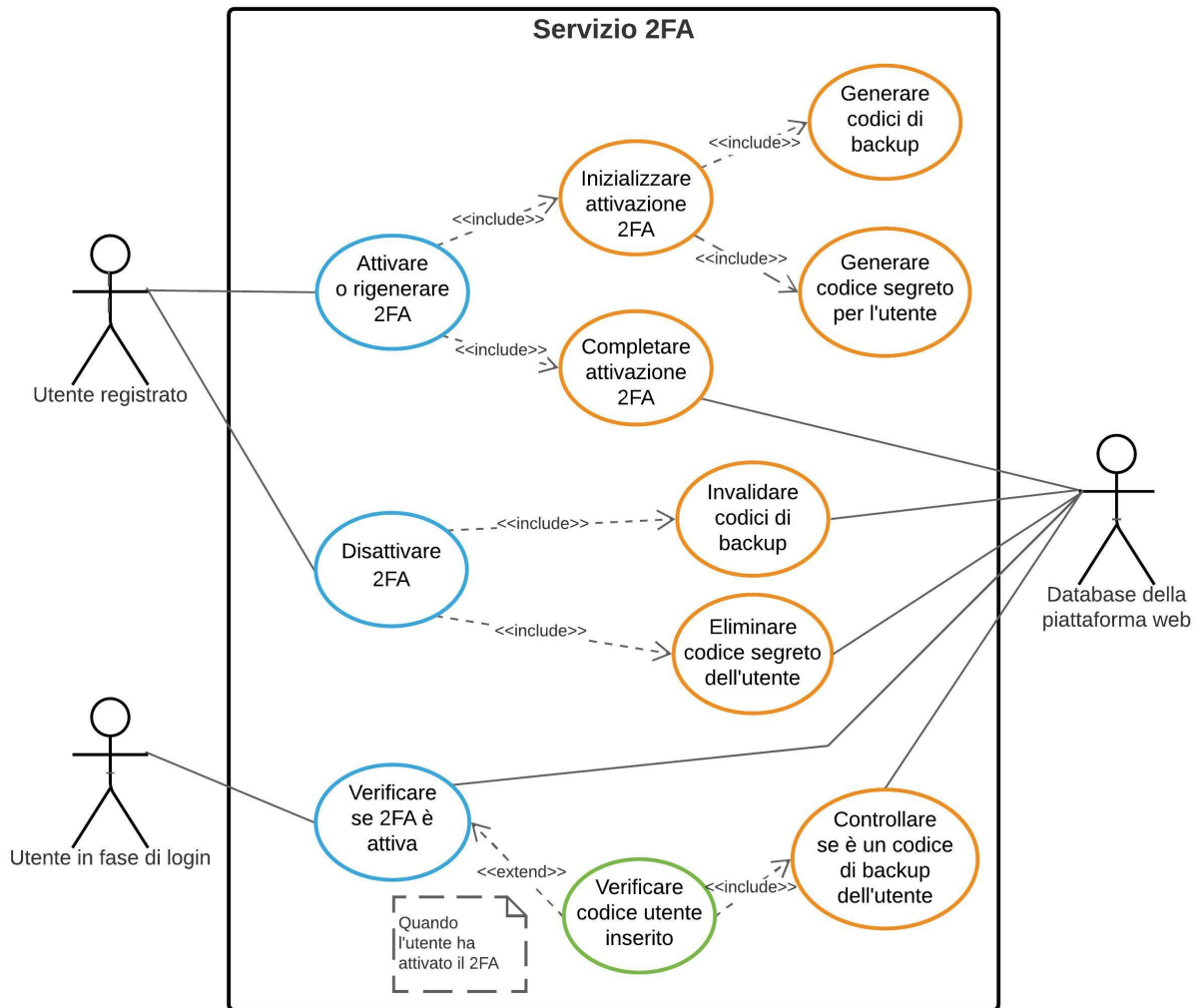


Figura 4.1: Diagramma dei casi d'uso (*Use Case Diagram*) raffigurante il servizio 2FA da progettare

Gli ellissi di colore azzurro descrivono le principali azioni (*Base Use Case*) che saranno svolte dal servizio *2FA* e si trovano in relazione con l'attore che richiederà quel tipo di attività.

Gli ellissi di colore arancione denotano gli *Included Use Case* e sono sempre correlati ad un *Base Use Case*: ogni volta che un *Base Use Case* viene eseguito, anche l'*Included Use Case* associato verrà eseguito.

Gli ellissi di colore verde, infine, rappresentano gli *Extended Use Case* e, come gli ellissi arancioni, sono sempre associati ad un *Base Use Case*. In questo caso, però, gli *Extended Use Case* verranno eseguiti solo in determinate condizioni e **non** tutte le volte che il *Base Use Case* viene eseguito. Per esempio, l'invalidazione dei codici di backup e l'eliminazione del codice segreto dell'utente saranno eseguiti ogni volta che si esegue la disattivazione del servizio. La verifica del codice utente inserito, invece, verrà svolta solo nel caso in cui l'utente abbia attivato l'autenticazione a due fattori.

4.5 Ricerca e analisi dei componenti del servizio

Una volta delineate in modo generico le principali azioni che il sistema dovrà svolgere, è necessario ridurre l'astrazione e procedere con un'analisi approfondita con lo scopo di individuare i componenti del servizio.

Dal punto di vista architetturale, è opportuno che le funzionalità messe a disposizione siano accessibili tramite un unico gestore (*TwoFaManager*) che nasconda i dettagli strutturali e implementativi sottostanti in modo da migliorare l'usabilità e la flessibilità del sistema. Il gestore dovrà permettere l'attivazione e la disattivazione dell'autenticazione a due fattori e, per far ciò, dovrà interagire con un componente ausiliario (*TwoFaAuxManager*) che fornirà tutte le funzionalità necessarie:

- Creazione dei codici di backup
- Invalidazione dei codici di backup
- Generazione del codice segreto
- Validazione del codice inserito dall'utente durante la fase di attivazione
- Elaborazione del contenuto del codice *QR*
- Salvataggio in *database* dei codici di backup e del codice segreto

I codici di backup non sono obbligatori per attivare con successo l'autenticazione a due fattori: sono stati introdotti per offrire all'utente un ulteriore metodo di autenticazione in caso di imprevisti. Sarà opportuno, quindi, definire un'entità separata (*TwoFaBackupCode*) che rappresenti i codici di backup. Verrà designato, inoltre, un componente (*TwoFaBackupCodeManager*) con il compito di fornire le operazioni necessarie per la creazione e l'invalidazione dei codici di backup. In questo modo, *TwoFaAuxManager* non dovrà gestire anche la logica delle operazioni riguardanti i codici di backup. Separando le responsabilità, quindi, il codice scritto nella fase implementativa sarà più chiaro e mantenibile.

Sorge però un problema: avendo separato l'attivazione del *2FA* in due fasi, è necessario fornire i dati generati dalla prima fase alla seconda. Ricapitolando, la prima fase dovrà generare i codici di backup, il codice segreto e il contenuto del codice *QR* che fornirà i dati all'*autenticatore* presente sul

dispositivo dell'utente. Se l'utente decide di scansionare il codice *QR* e di inserire il codice temporaneo generato dall'autenticatore, significa che conferma la propria intenzione di attivare il servizio. I dati generati, dunque, dovranno essere trasferiti alla seconda fase affinché possano essere inseriti in *database*. È necessario, pertanto, che i dati prodotti vengano salvati soltanto temporaneamente per permettere alla seconda fase di recuperarli in caso di conferma da parte dell'utente.

Un buon approccio al problema prevede di salvare i dati nella *cache* del sistema esterno in cui viene importato il servizio di autenticazione. Molte piattaforme web, tuttavia, hanno una struttura decentralizzata e piuttosto complessa che prevede l'utilizzo di più di un *database*. Ciò richiederebbe la creazione di un server apposito, accessibile da tutti i componenti del sistema, in cui salvare i dati temporanei. Al giorno d'oggi, però, esistono molte tecnologie che offrono servizi di *Cloud Storage* che presentano numerosi vantaggi rispetto alla realizzazione da zero di un server dedicato:

- **Risparmio economico.** Non sono necessari alti investimenti iniziali e costi di infrastruttura: l'hardware e tutto il necessario è a carico del *provider*. È possibile, dunque, utilizzare fin da subito le risorse, il *networking* e le soluzioni di sicurezza. In questo modo, dunque, non ci si deve più preoccupare degli aspetti infrastrutturali e di manutenzione.
- **Disponibilità.** Le risorse sono sempre disponibili e accessibili da remoto in qualunque luogo e in qualunque momento.
- **Pay-per-use.** Al cliente verranno addebitate solo le risorse effettivamente utilizzate. Ciò permette di raggiungere il miglior compromesso tra prestazione e costo.
- **Scalabilità.** Il *Cloud* offre un'elevata scalabilità, mantenendo il servizio funzionante anche in caso di rapidi incrementi/decrementi di risorse.
- **Sicurezza del sistema.** Vengono utilizzate procedure tecnologiche di alto livello per proteggere al meglio il sistema e i dati ivi contenuti (es. procedure di *backup* e *recovery*).

I servizi *Cloud*, però, non potranno mai garantire lo stesso livello di riservatezza dei dati di un *server* fisico accessibile soltanto in una rete privata aziendale. Un altro svantaggio è la connessione internet: senza, infatti, non sarà possibile eseguire alcuna operazione. È necessario, pertanto, valutare sempre il rapporto tra rischi e benefici.

Per quanto riguarda il servizio in fase di progettazione, è fondamentale avere costi contenuti e tempistiche di inizializzazione molto basse rispetto ad un alto livello di riservatezza. In fin dei conti si tratta di un servizio di *cache*: i dati verranno salvati soltanto per un breve periodo di tempo e saranno eliminati una volta completata la procedura di attivazione del *2FA* (ovvero quando l'utente inserisce un codice temporaneo di conferma valido). È ragionevole, dunque, utilizzare i servizi di *Cloud Storage* e, pertanto, sarà necessario introdurre un componente aggiuntivo (*TwoFaCacheManager*) che dovrà permettere di salvare, recuperare ed eliminare i dati in *cache*.

4.5.1 SQL vs NoSQL

I servizi di *Cloud Storage* forniscono due tipologie di *database*: *SQL* e *NoSQL*. *Structured Query Language* (*SQL*) è un linguaggio volto all'interrogazione di dati altamente strutturati. Quest'ultimi vengono

rappresentati da un insieme di tuple organizzate in relazioni: in questo modo è possibile astrarre la reale rappresentazione fisica dei dati. I punti di forza (nonché alcuni dei maggiori vincoli) sono le transazioni (in genere assenti nei *database* non relazionali) e la normalizzazione dei dati. Le transazioni, per essere considerate tali, devono rispettare le proprietà *ACID* (*Atomicity, Consistency, Isolation, Durability*) che, dovendo essere soddisfatte ad ogni modifica del *database*, comportano una notevole rigidità. Lo scopo della normalizzazione, invece, è di ridurre la ridondanza dei dati e permettere di effettuare controlli di integrità su quest'ultimi. Il grande svantaggio, però, sono le operazioni di *join*, necessarie a combinare le tuple di due o più relazioni. Il *join*, infatti, comporta la creazione di una tabella di relazione per poter collegare una certa entità alle entità ad essa associate. Nella controparte *NoSQL*, invece, è possibile definire entità annidate che consentono di eliminare l'*overhead* causato dalla creazione di nuove tabelle di *join*.

I *database NoSQL* (*Not Only SQL*), d'altro canto, nascono con lo scopo di rappresentare e gestire dati eterogenei (non strutturati). Questo tipo di *database*, infatti, è caratterizzato da schemi dinamici che permettono ad ogni dato di avere una propria struttura. La gestione di dati non strutturati, però, non consente la definizione di una sintassi univoca: la modalità con cui gestire i dati cambia da *database* a *database* (chiave-valore, documenti *JSON*, grafi, in memoria, ...). Ciò rende i *database NoSQL* estremamente versatili e adatti alla gestione di dati che tendono ad evolvere nel tempo. Tuttavia, dovendo gestire dati non normalizzati, i *database NoSQL* sono caratterizzati da alcune problematiche non trascurabili, prima fra tutte la duplicazione. Nonostante sia vantaggiosa per quanto riguarda la velocità di accesso, la duplicazione comporta un aumento dello spazio su disco richiesto dal *database* e mette a rischio l'integrità dei dati. In caso di modifica di un dato duplicato, è necessario aggiornare ogni singola copia del dato. I *database NoSQL*, infine, offrono la possibilità di scalare orizzontalmente (*sharding*), migliorando sia le *performance* che la capacità totale del *database*.

È fondamentale sottolineare, però, che non esiste un vincitore assoluto tra *SQL* e *NoSQL*: esiste solo il più adatto ad una determinata circostanza. Il servizio in fase di progettazione deve simulare il comportamento di una *cache* e, di conseguenza, i *database NoSQL* risultano la scelta più opportuna in quanto offrono elevate prestazioni e una notevole flessibilità. In una *cache*, infatti, le operazioni sui dati saranno molto frequenti: garantire buone *performance* è fondamentale.

4.5.2 Eliminazione automatizzata dei dati in cache

Al momento, però, i dati in *cache* verranno eliminati solo se l'utente conferma la propria decisione di attivare il *2FA* inserendo un codice temporaneo ricavato da un *authenticator* sul proprio telefono. Di conseguenza, se l'utente decide di non confermare la sua scelta (per un qualsiasi motivo), soltanto la prima fase dell'attivazione del servizio verrà eseguita e i dati rimarranno salvati indefinitamente in *cache*.

Una soluzione al precedente problema consiste nel salvare i dati generati dalla prima fase in *cache* per un periodo limitato. Il servizio di *cloud storage* scelto (*Firestore*), tuttavia, non fornisce quest'opzione (a differenza dei sistemi di caching standard). Per ovviare a ciò, si potrebbe inviare un messaggio ogniqualvolta la prima fase di attivazione del *2FA* viene completata. Il messaggio, successivamente, verrà recapitato ad un apposito componente che avrà la responsabilità di decidere se i dati in *cache* dovranno essere eliminati o meno.

Per far ciò, il messaggio dovrà contenere il nome del documento salvato in *cache* (per individuare ed eliminare i dati correlati) e la data e l'ora in cui i dati sono stati salvati. Il servizio di messaggistica, inoltre, dovrà inviare il messaggio finché non riceverà un *acknowledgement* (un messaggio che attesta la corretta ricezione ed elaborazione del precedente messaggio) da parte del componente ricevente.

Per quanto riguarda la permanenza dei dati in *cache*, si è scelto di adottare un intervallo di 10 minuti: una volta trascorso questo periodo si dovrà procedere con la loro eliminazione. Per evitare, inoltre, che il sistema venga sommerso di messaggi dopo la mancata ricezione di un *acknowledgement*, è necessario definire un ritardo (*backoff*) di 10 minuti in caso di mancato *acknowledgement*: in questo modo il sistema invierà immediatamente soltanto un messaggio, per poi inviarne un secondo dopo 10 minuti.

Sul mercato sono presenti diversi servizi di messaggistica che offrono queste funzionalità. Si dividono principalmente in due modelli che differiscono in base a come i dati vengono trasferiti dai mittenti ai destinatari: **code di messaggi** e **messaggistica *publish-subscribe* (*pub/sub*)**.

La messaggistica *publish-subscribe* permette l'invio di ogni messaggio presente in un *topic* a più di un *consumer* (destinatario) e, inoltre, assicura che ogni *consumer* riceva i messaggi di un *topic* esattamente nell'ordine con cui il sistema di messaggistica li ha ricevuti.

Per quanto riguarda il servizio da progettare, la coda di messaggi è il modello di messaggistica che si adatta meglio in quanto l'ordine con cui verranno eliminati i dati in *cache* non è importante e non vi è una moltitudine di *consumer* che necessita di processare ogni messaggio di un determinato *topic*. Tuttavia, dato che le piattaforme web dell'azienda non utilizzano questo modello, è poco sensato progettare da zero questa tipologia di servizio. L'approccio che si è deciso di seguire, dunque, è quello del *publish-subscribe* utilizzando il servizio *Google Cloud Pub/Sub*. Nonostante, quindi, non siano necessarie tutte le funzionalità messe a disposizione da questo modello, il risparmio di tempo e costi che ne deriva ha fatto propendere la scelta per questo servizio.

Google Cloud Pub/Sub (offerto dalla *Google Cloud Platform*) è un servizio di messaggistica in tempo reale completamente gestito che permette ai servizi di comunicare in maniera asincrona ed indipendente. Consente di creare sistemi di produttori (*publisher*) e consumatori (*subscriber*) di eventi. I *publisher* inviano eventi sotto forma di messaggi al servizio *Pub/Sub* senza sapere come o quando questi saranno processati. Successivamente *Pub/Sub* consegna gli eventi a tutti i servizi interessati (in modo che possano reagire all'evento). Il funzionamento di *Google Cloud Pub/Sub* è illustrato in Figura 4.2:

1. Un servizio *publisher* invia un messaggio ad un certo *topic* (una specifica coda di messaggi pubblicati dai *publisher*)
2. I messaggi vengono salvati in una memoria separata (*Message Storage*) finché non verrà ricevuto un messaggio di *acknowledgement* da parte di tutti i servizi *subscriber*
3. Per ricevere i messaggi, i *subscriber* creano una *subscription* al *topic* in questione (rappresenta il flusso dei messaggi provenienti da un *topic* specifico da consegnare ai servizi *subscriber*)
4. I *subscriber* possono ricevere i messaggi in due modi: *Pub/Sub* si occupa di inviarli (*push*) all'*endpoint* scelto dal *subscriber* oppure il *subscriber* stesso può decidere di ritirare (*pull*) i messaggi da *Pub/Sub*

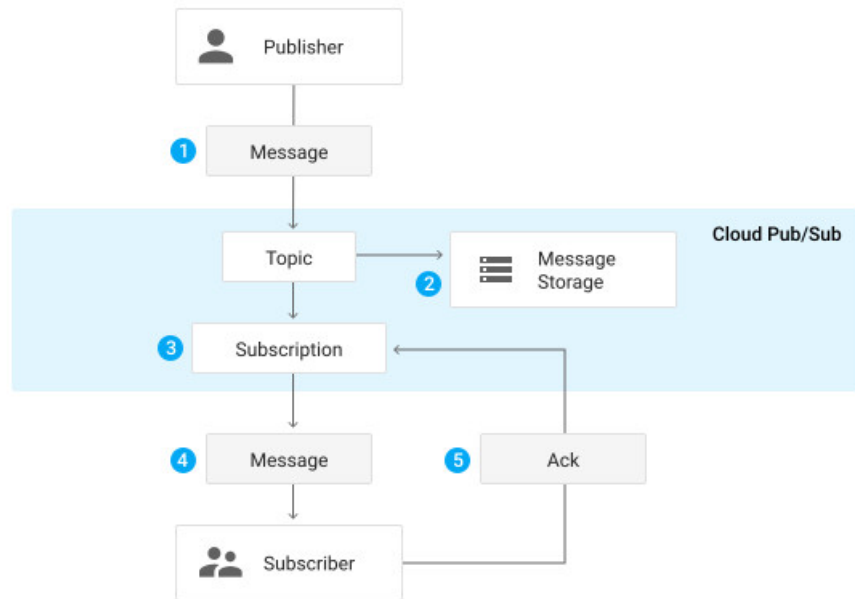


Figura 4.2: Schema raffigurante il funzionamento di *Google Cloud Pub/Sub* (da [4])

5. I *subscriber* inviano un messaggio di *acknowledgement* alla *subscription* di *Pub/Sub*. In questo modo il messaggio può essere rimosso dal *backlog* della *subscription* e, di conseguenza, non verrà più consegnato.

I *subscriber* che utilizzano il metodo *pull* per ricevere i messaggi (*pull subscribers*) devono poter effettuare richieste *HTTPS* (*Hypertext Transfer Protocol Secure*) alle *API* di *Google* per ricavare i messaggi dalle *subscription* di *Pub/Sub*. I *push subscriber*, invece, devono fornire un *endpoint* valido che possa accettare richieste *HTTPS* di tipo *POST*.

L'adozione di *Google Cloud Pub/Sub*, dunque, porterà molteplici vantaggi, tra cui:

- Riduzione delle dipendenze tra i servizi: un servizio non è obbligato a conoscere i cambiamenti che avvengono in un altro servizio. I servizi, infatti, risultano indipendenti tra loro, rendendo l'architettura dell'applicazione più flessibile.
- Non è necessario adottare delle contromisure per arginare il *Single Point Of Failure*, ovvero una parte del sistema il cui malfunzionamento può causare delle situazioni inconsistenti oppure può portare alla cessazione del servizio offerto dal sistema.
- Elevata scalabilità e una considerevole robustezza in caso di imprevisti.

Tornando al progetto, sarà necessaria la creazione di un componente che gestirà l'invio dei messaggi (*TwoFaMessageManager*). Questo componente sarà richiamato ogni volta che verrà eseguita la prima fase di attivazione del *2FA*.

Il messaggio, come detto precedentemente, dovrà contenere il nome del documento in cui sono stati salvati tutti i dati essenziali all'attivazione del *2FA* e verrà inviato ad uno specifico *topic* su *Google Cloud Pub/Sub*, che lo instraderà a tutti i *subscriber* iscritti a quel determinato *topic*. Per poter

ricevere il messaggio, è necessario creare una *subscription* ad hoc e selezionare *push* come metodo di consegna dei messaggi (*Google Cloud Pub/Sub*, quindi, invierà il messaggio su un opportuno *endpoint*). La *subscription*, inoltre, dovrà attuare una *retry policy* che imposterà un *backoff* di 10 minuti prima di rispedire un messaggio se non verrà ricevuto un *acknowledgement* entro 30 secondi dall'invio del messaggio.

Sarà necessaria, quindi, l'introduzione di un componente (*TwoFaCleanupperController*) che si occuperà di ricevere i messaggi indirizzati a quello specifico *endpoint* e che gestirà la logica riguardante l'eliminazione dei dati in *cache*.

4.6 Attivazione e disattivazione del 2FA

Dopo aver definito i principali componenti che costituiranno il servizio, è opportuno analizzare e descrivere come quest'ultimi interagiranno tra loro per consentire l'attivazione e la disattivazione dell'autenticazione a due fattori.

L'autenticazione, invece, viene gestita dalla libreria *2FA* adottata (*schneb/2fa*). Nel prossimo capitolo verrà spiegato in dettaglio l'installazione di questa libreria e come verrà gestita l'autenticazione.

4.6.1 Attivazione

L'attivazione è suddivisa in due fasi: la prima si occuperà di generare tutti i dati necessari all'attivazione mentre la seconda dovrà abilitare il *2FA* per l'utente in questione utilizzando i dati generati dalla prima fase. Entrambe le fasi dovranno utilizzare le funzionalità offerte dalla libreria *schneb/2fa* per completare i singoli compiti.

La seconda fase, tuttavia, verrà eseguita soltanto se l'utente confermerà la propria intenzione di attivare il *2FA* inserendo un codice temporaneo valido. Sarà compito del sistema esterno mostrare il codice *QR* (in modo che l'utente lo possa scansionare per condividere i dati con l'applicazione *mobile*) e gestire l'inserimento del codice da parte dell'utente.

Il diagramma di sequenza di Figura 4.3 permette di visualizzare graficamente in che modo e in quale ordine i componenti del sistema interagiscono tra loro per permettere l'attivazione dell'autenticazione a due fattori. Consente, peraltro, di riassumere il funzionamento del sistema finora progettato.

Prima fase

Il componente *TwoFaManager* dovrà mettere a disposizione una procedura per eseguire la fase di inizializzazione necessaria all'attivazione del *2FA*. Per far ciò, *TwoFaManager* dovrà interagire con *TwoFaAuxManager* per:

- generare i codici di backup da fornire all'utente
- creare il codice segreto da associare all'utente registrato nel sistema
- elaborare il contenuto del codice *QR* da mostrare all'utente
- salvare tutti i dati generati precedentemente in *cache*

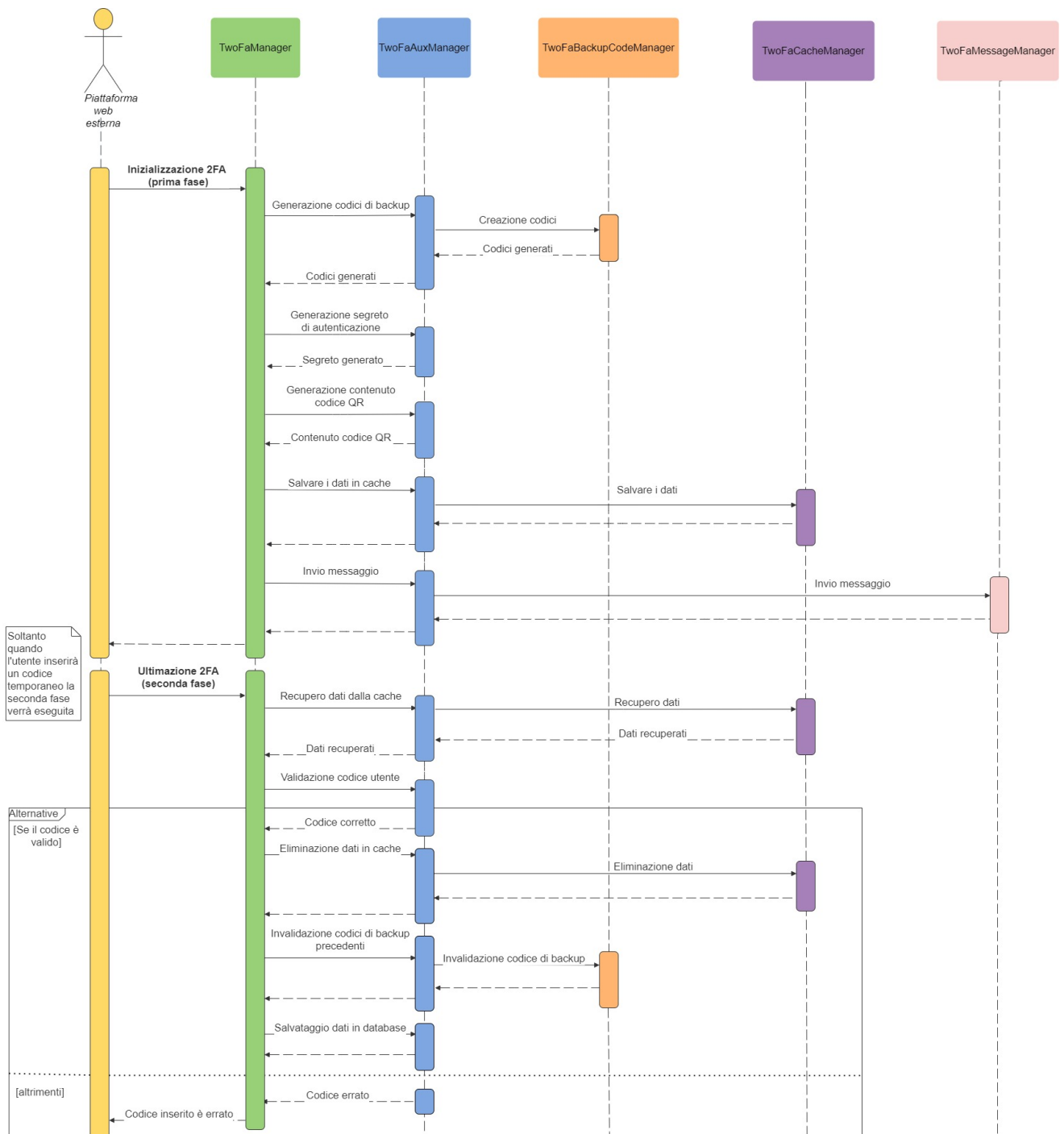


Figura 4.3: Diagramma di sequenza (*Sequence Diagram*) che illustra l'ordine delle interazioni necessarie all'attivazione del 2FA.

- inviare un messaggio a *Google Cloud Pub/Sub* per automatizzare l'eliminazione dei dati in *cache*

TwoFaAuxManager, a sua volta, dovrà far riferimento a:

- *TwoFaBackupCodeManager* per generare i codici di backup
- *TwoFaCacheManager* per salvare i dati generati in *cache*
- *TwoFaMessageManager* per inviare il messaggio a *Google Cloud Pub/Sub*

Google Cloud Pub/Sub, una volta ricevuto il messaggio, lo trasmetterà ad un determinato *endpoint* della piattaforma web gestito dal componente *TwoFaCleanupperController*. Sarà compito di quest'ultimo, dunque, gestire l'eliminazione dei dati in *cache*.

Seconda fase

Nel momento in cui il codice viene inserito e confermato dall'utente, la seconda fase (messa a disposizione da *TwoFaManager*) verrà eseguita. Per prima cosa sarà necessario recuperare il documento in *cache* contenente i dati generati dalla prima fase. Per far ciò, bisognerà fornire il nome del documento alla seconda fase e, successivamente, interagire con *TwoFaAuxManager* (che a sua volta interagirà con *TwoFaCacheManager*).

In seguito, si farà riferimento a *TwoFaAuxManager* per validare il codice inserito dall'utente. Nel caso in cui il codice sia corretto, si dovrà:

- eliminare il documento in *cache*
- invalidare i codici di backup precedentemente assegnati all'utente
- salvare tutti i dati e le modifiche effettuate fino a questo momento nel *database*

Per eliminare il documento e invalidare i codici di backup si dovrà far riferimento a *TwoFaCacheManager* e *TwoFaBackupCodeManager*. Il salvataggio in *database*, infine, sarà gestito direttamente da *TwoFaAuxManager*.

4.6.2 Disattivazione

TwoFaManager, oltre a gestire l'attivazione, dovrà anche fornire una procedura per disattivare l'autenticazione a due fattori per un utente specifico. Per completare la disattivazione, dovranno essere invalidati i codici di backup associati all'utente (facendo riferimento a *TwoFaAuxManager*) e, successivamente, sarà necessario eliminare il codice segreto di autenticazione associato all'utente condiviso tra il server e il dispositivo *mobile*.

4.7 Dispositivi mobile

L'ottavo requisito non funzionale elencato nel Capitolo 3 dichiara che il servizio dovrà essere utilizzabile anche da applicazioni sviluppate per dispositivi *mobile*. Sarà necessario, quindi, progettare e sviluppare una *Web API* che metta a disposizione le principali funzionalità offerte dal servizio:

- attivazione del *2FA* (prima e seconda fase)
- disattivazione del *2FA*
- validazione del codice inserito dall'utente in fase di autenticazione nel caso in cui il *2FA* sia attivo

Per far ciò, si dovrà introdurre un ulteriore componente (*TwoFaController*) che, interagendo con *TwoFaManager*, gestirà la *Web API*.

Per quanto riguarda l'attivazione, l'*API* dovrà inviare anche una *e-mail* all'utente interessato contenente il codice *QR* e i codici di backup. L'invio della *e-mail* è necessario in quanto l'utente scansiona il codice *QR* tramite la fotocamera del proprio cellulare per consentire il passaggio dei dati all'*authenticator* installato sul proprio dispositivo. Ritrovandosi il codice *QR* sul cellulare, l'utente non avrebbe modo di condividere i dati con l'*authenticator* se non inserendo manualmente una stringa contenente tutti i parametri necessari. L'*e-mail*, quindi, consente all'utente di poter visualizzare il codice *QR* anche su altri dispositivi permettendo, in questo modo, la scansione del codice tramite il proprio telefono.

4.7.1 Web API

Un'*API* (*Application Programming Interface*) è un insieme di protocolli e definizioni che permettono lo scambio di informazioni tra componenti *software*. Ci sono diversi tipi di *API* ma la tipologia più adatta al progetto è quella delle *Web API* in quanto consentono:

- a più applicazioni web di comunicare tra loro
- a più parti di un'applicazione web di comunicare tra loro
- a un'applicazione client esterna di comunicare con un'applicazione web

L'ultimo punto è proprio quello di cui il progetto necessita. L'azienda *Cogito srl* possiede numerose piattaforme web che dispongono sia di un sito web accessibile tramite *browser* che di applicazioni per *smartphone* per diversi sistemi operativi (*Android* e *iOS*). Tutte queste applicazioni devono poter offrire le stesse funzionalità o aver accesso agli stessi contenuti. Senza una *Web API*, ogni applicazione *client* andrebbe a definire la propria logica in un modo specifico e, molto spesso, diverso dagli altri *client*. L'aggiunta di nuove funzionalità ad una piattaforma web con questo tipo di architettura potrebbe diventare rapidamente molto complesso, dovendo implementare le nuove funzionalità su ogni applicazione *client*. Questo tipo di approccio, dunque, può causare l'insorgere di numerosi *bug* e ritardi nello sviluppo.

L'utilizzo di una *Web API*, invece, semplifica notevolmente il processo offrendo a tutte le applicazioni *client* un'interfaccia comune tramite cui interagire con la logica del servizio erogato dalla piattaforma web e con il relativo *database*. Ciò permette alle applicazioni di utilizzare le funzionalità della piattaforma web senza conoscere come vengono effettivamente implementate.

Una *Web API* dovrebbe essere progettata per consentire l'indipendenza dalla piattaforma e l'evoluzione del servizio. L'*API*, quindi, deve poter evolversi nel tempo indipendentemente dalle applicazioni *client* che, a loro volta, devono poter richiamare l'*API* a prescindere da come è stata implementata internamente: è necessario l'utilizzo di protocolli standard e un meccanismo che consenta al *client* e alla piattaforma web di concordare il formato dei dati da scambiare.

Un'API, infatti, può essere concepita come una forma di contratto: inviando una richiesta strutturata in un certo modo si otterrà una risposta che segue una struttura determinata. In genere, le *Web API* utilizzano *HTTP* (*HyperText Transfer Protocol*) per richiedere messaggi e fornire una definizione della struttura dei messaggi di risposta che, di solito, assumono la forma di file *XML* (*eXtensible Markup Language*) o *JSON* (*JavaScript Object Notation*).

Le *Web API*, solitamente, adottano uno di questi due approcci: *REST* (*REpresentational State Transfer*) o *SOAP* (*Simple object access protocol*). Si tratta di due diversi approcci alla trasmissione dei dati. Il primo (*REST*) è un insieme di principi architetturali e linee guida mentre il secondo (*SOAP*) è un protocollo ufficiale gestito dal *W3C* (*World Wide Web Consortium*).

Le prossime sezioni confronteranno questi due approcci in modo tale da selezionare quello più adatto alle esigenze del progetto.

4.7.2 SOAP

SOAP è un protocollo che consente lo scambio di messaggi tra componenti software. Si basa sul meta-linguaggio *XML* e venne ideato con l'obiettivo di consentire la comunicazione tra applicazioni realizzate con linguaggi e piattaforme differenti.

Trattandosi di un protocollo, viene richiesto il rispetto di regole specifiche che tendono, però, a renderne più complesso l'utilizzo. È necessaria, infatti, la creazione di un file *WSDL* (*Web Services Description Language*) in formato *XML* tramite il quale descrivere l'interfaccia pubblica di un servizio web. Il file dovrà contenere informazioni su:

- operazioni messe a disposizione dal servizio
- protocollo di comunicazione da utilizzare per accedere al servizio
- formato dei messaggi accettati in input e restituiti in output dal servizio
- *endpoint* del servizio (ovvero un indirizzo in formato *URI* che specifica dove trovare il servizio in rete)

Il protocollo, inoltre, è indipendente dalla modalità di trasporto ma, generalmente, le tipologie più usate sono *HTTP* (per permettere il suo utilizzo con l'infrastruttura Internet esistente) e *SMTP*.

SOAP, però, presenta alcuni importanti svantaggi. L'utilizzo di *XML* per strutturare i messaggi e il rispetto delle regole imposte dal protocollo causano un notevole incremento della dimensione del messaggio. Quest'ultimo, infatti, contiene un'elevata quantità di dati e ciò provoca un consumo di *bandwidth* molto superiore rispetto al protocollo *REST*.

Un altro punto debole è il file *WSDL*: trattandosi di una forma di contratto tra il client e il server, ogni cambiamento apportato a questo file può comportare numerose modifiche alle applicazioni *client*.

SOAP, infine, supporta solo il linguaggio *XML* per quanto riguarda la formattazione dei messaggi e, oltretutto, non consente di memorizzare le precedenti richieste nella *cache* del *browser* (non sarà possibile accedere in un secondo momento ad una richiesta senza rinviarla all'API).

4.7.3 REST

REST è un approccio architetturale proposto da Roy Fielding riguardante la progettazione di servizi web. Non trattandosi di un protocollo o di uno standard, i principi suggeriti da *REST* sono implementabili in diversi modi.

Questa tipologia di approccio è indipendente da qualsiasi protocollo sottostante ma la maggior parte delle implementazioni di una *REST API* utilizzano *HTTP* come protocollo applicativo. Ciò è dovuto al fatto che *REST* su *HTTP* consente l'utilizzo di standard aperti e permette di non vincolare l'implementazione dell'*API* o delle applicazioni *client*: può essere utilizzato qualsiasi linguaggio che può generare richieste *HTTP* e analizzare risposte *HTTP*. *REST*, inoltre, non vincola il formato dei messaggi da trasferire: è possibile utilizzare una vasta varietà di linguaggi (*HTML*, *XML*, *JSON*, *plain text*, ecc.). Non è necessario neanche l'utilizzo di una *bandwidth* elevata in quanto i messaggi, essendo solitamente scritti in *JSON*, mantengono dimensioni relativamente contenute. Le *API* che seguono questo approccio, dunque, risultano molto flessibili e facilmente configurabili.

Le *REST API* sono progettate in base a risorse¹. Ogni risorsa dispone di un *URI* (*Uniform Resource Identifier*) che la identifica in modo univoco. Un'*API* viene considerata *RESTful* se rispetta i seguenti principi architetturali:

1. Architettura *client-server* costituita da *client*, *server* e risorse
2. Comunicazione *client-server stateless*: nessuna informazione del *client* viene salvata sul *server* tra una richiesta e l'altra. Ogni richiesta è indipendente dalle altre e può essere eseguita in qualsiasi ordine
3. *Caching* dei dati (rendendo superflue alcune interazioni tra *client* e *server*)
4. Interfaccia uniforme per i componenti che permette di trasferire i dati in un formato standard e non in base alle esigenze di un'applicazione specifica
5. Le interazioni *client-server* possono essere mediate da livelli aggiuntivi che possono offrire altre funzionalità (bilanciamento carico, condivisione *cache*, ecc.) permettendo, quindi, di recuperare le informazioni richieste in un modello gerarchico. Tuttavia, nè il *client* nè il *server* devono essere a conoscenza se stanno comunicando con l'applicazione finale o con un intermediario
6. I *server* possono ampliare le funzionalità del *client* inviando codice eseguibile quando richiesto (opzionale)

Per le *REST API* basate su *HTTP*, l'interfaccia uniforme include l'uso dei metodi *HTTP* standard (*GET*, *POST*, *PUT*, *PATCH* e *DELETE*) per l'esecuzione di operazioni sulle risorse.

Per soddisfare il quarto requisito, inoltre, Roy Fielding indica quattro vincoli che devono essere rispettati:

1. **Identificazione delle risorse:** stabilire un meccanismo per identificare le risorse accessibili sul sistema

¹una risorsa è un qualsiasi oggetto, servizio, dato accessibile dal *client*

2. **Manipolazione delle risorse tramite una loro rappresentazione:** le risorse non vengono accedute direttamente ma vengono fornite delle rappresentazioni
3. **Messaggi autodescrittivi:** la semantica delle risorse e dei metadati deve essere accessibile a tutti i componenti del sistema
4. **Hypermedia come il motore per la gestione dello stato dell'applicazione:** ogni risorsa contribuisce alla rappresentazione dello stato dell'applicazione e deve fornire l'accesso ad eventuali risorse collegate

L'ultimo vincolo stabilisce che quando un *client* richiede una risorsa al *server*, quest'ultimo deve restituire la risorsa (una sua rappresentazione) con i collegamenti ad altre risorse correlate. Il *client*, in questo modo, può navigare tra le risorse disponibili tramite i reciproci collegamenti. Si tratta, quindi, di indirizzi che vengono scoperti gradualmente accedendo alla risorse messe a disposizione del *server* e non di indirizzi che il *client* conosce già dall'inizio. Questa modalità di interazione non è altro che la navigazione di un *browser* tra le pagine web.

Un vero *client REST*, quindi, dovrebbe essere a conoscenza solo dell'*entry point* dell'*API*. Questo significa che il *server*, in un sistema che implementa i principi *REST*, ha un ruolo predominante nello stabilire il comportamento del *client* (avendo la possibilità di cambiare gli *URI* senza causare alcun tipo di problema). Quest'ultimo, infatti, dovrà solo interpretare la rappresentazione delle risorse e seguire i percorsi ammessi dai collegamenti tra una risorsa e l'altra.

In realtà, quindi, la maggior parte delle *API* presenti sul mercato non sono vere *REST API*. La progettazione di una *REST API*, al giorno d'oggi, consiste principalmente nel definire delle risorse accessibili via web, identificarle con degli *URL* e mappare le operazioni *CRUD* (*Create, Retrieve, Update, Delete*) su tali risorse ai metodi del protocollo *HTTP*. Per questo motivo, Leonard Richardson (2008) ha proposto per le *Web API* il seguente modello di maturità:

- **Livello 0:** definizione di un *URI* e operazioni interamente costituite da richieste *POST* a tale *URI* (utilizzare *HTTP* solo come sistema di trasporto per interazioni remote)
- **Livello 1:** creazione di *URI* separati per le singole risorse
- **Livello 2:** uso dei metodi *HTTP* per definire le operazioni sulle risorse e introduzione dei codici di risposta *HTTP*.
- **Livello 3:** uso di *hypermedia*

Il precedente modello non è da intendersi come una definizione dei livelli di *REST*. Secondo Roy Fielding, infatti, il livello 3 non coincide con un'*API* realmente *RESTful* ma ne costituisce soltanto una precondizione. Nonostante ciò, il modello di Richardson è uno strumento che permette di comprendere più a fondo le idee e i concetti che stanno alla base dell'approccio *REST*.

4.7.4 REST vs SOAP

Nonostante una *REST API* debba aderire a tutti i principi descritti precedentemente, l'implementazione risulta più agevole e flessibile se confrontata con un protocollo prefissato come *SOAP*. Una *REST API* dovrebbe essere utilizzata quando:

- la *bandwidth* è un vincolo. I messaggi *SOAP*, infatti, consumano una larghezza di banda più elevata (in quanto hanno dimensioni nettamente maggiori)
- non c'è bisogno di mantenere uno stato di informazioni tra una richiesta e l'altra. Se, invece, c'è bisogno di utilizzare un dato di una richiesta precedente, è necessario utilizzare *SOAP*
- si deve ricorrere necessariamente al *caching* (inserire molte richieste in *cache*)
- è richiesto un approccio veloce ed efficace (l'implementazione di servizi *REST* è molto più agevole rispetto a quella dei servizi *SOAP*)

Una *SOAP API*, invece, dovrebbe essere usata quando:

- bisogna garantire un certo livello di affidabilità e sicurezza
- il *client* ed il *server* si sono accordati sul formato con cui scambiare i dati (*SOAP* fornisce delle specifiche rigide per questo tipo di interazioni)
- lo stato deve essere mantenuto tra una richiesta e l'altra

Tornando al servizio in fase di progettazione, dato il poco tempo a disposizione è necessario utilizzare un approccio che consenta un'implementazione flessibile e veloce. Oltre a ciò, la maggior parte delle piattaforme web sviluppate dell'azienda *Cogito srl* utilizzano *REST API* e adottano un approccio *stateless* per quanto riguarda la comunicazione tra *client* e *server* (utilizzando per esempio i *JSON Web Token* in fase di autenticazione).

Sembrerebbe, dunque, opportuno adottare l'approccio *REST*. Sorge però un problema: affinché il *2FA* sia utilizzabile tramite *API*, quest'ultima deve essere di tipo *stateful*. Una sessione, infatti, è necessaria per salvare lo stato del login (per sapere se l'utente ha già completato l'autenticazione a due fattori o no). Sarà necessario, quindi, creare una sessione che verrà passata dal *client* ad ogni chiamata.

In conclusione, dopo aver analizzato il problema, si è deciso di adottare comunque l'approccio *REST* per tutti i vantaggi elencati precedentemente. L'*API* risultante, però, dovrà essere di tipo *stateful* (perciò non potrà essere *RESTful*) ma continuerà a seguire i restanti principi *REST*. Verrà adottata, quindi, una tipologia di *Web API* parzialmente ispirata alle linee guida di *REST*.

4.8 Diagramma dei componenti

Nelle precedenti sezioni sono stati delineati ed analizzati i principali componenti che andranno a costituire il servizio di autenticazione a due fattori. È opportuno, quindi, concludere la fase di progettazione riempiendo le responsabilità e le dipendenze dei singoli componenti.

A tal fine è stato realizzato il diagramma di Figura 4.4: lo schema raffigurato prende spunto dal diagramma delle classi *UML* (*Unified Modeling Language*) e permette di descrivere la struttura interna del servizio. Il diagramma, infatti, consente di osservare le dipendenze tra i vari componenti del servizio e il comportamento di ogni elemento (ovvero tutte quelle funzionalità che un componente deve fornire). Lo schema, però, illustra il servizio ad un alto livello di astrazione, mostrando solo i componenti individuati in questo capitolo e non tutti gli elementi che andranno a comporre il servizio. La struttura interna completa, infatti, sarà descritta nel prossimo capitolo.

TwoFaManager sarà l'unico componente con cui i sistemi esterni dovranno interagire per utilizzare le funzionalità del servizio (*entry point*). Il componente, infatti, dovrà fornire dei metodi per consentire l'attivazione e la disattivazione del *2FA* per uno specifico utente. Per far ciò, *TwoFaManager* dovrà far riferimento a *TwoFaAuxManager*, un componente ausiliario che metterà a disposizione tutte le funzionalità necessarie. Sarà, quindi, compito di *TwoFaAuxManager* interagire con i singoli componenti del servizio.

Oltre ai componenti precedentemente descritti, lo schema mostra anche il componente *TwoFaController* e due entità che faranno parte del servizio (*TwoFaCleanupMessage* e *TwoFaBackupCode*). *TwoFaController*, utilizzando le funzionalità messe a disposizione da *TwoFaManager*, gestirà la *REST API* che permetterà al servizio di essere usufruito anche dalle applicazioni *mobile*.

TwoFaCleanupMessage, invece, rappresenta il messaggio che verrà inviato a *Google Cloud Pub/Sub* ogniqualvolta verrà eseguita la prima fase dell'attivazione del *2FA*. Il messaggio, successivamente, verrà spedito da *Google Cloud Pub/Sub* a *TwoFaCleanupperController* che si occuperà dell'eliminazione dei dati salvati in *cache*. Sarà necessario, dunque, inserire nel messaggio il nome del documento salvato in *cache* e la data e l'ora in cui è avvenuto il salvataggio.

TwoFaBackupCode, infine, rappresenta i codici di backup da fornire all'utente durante l'attivazione del *2FA*. L'entità dovrà memorizzare il codice, la validità di quest'ultimo e l'utente associato.

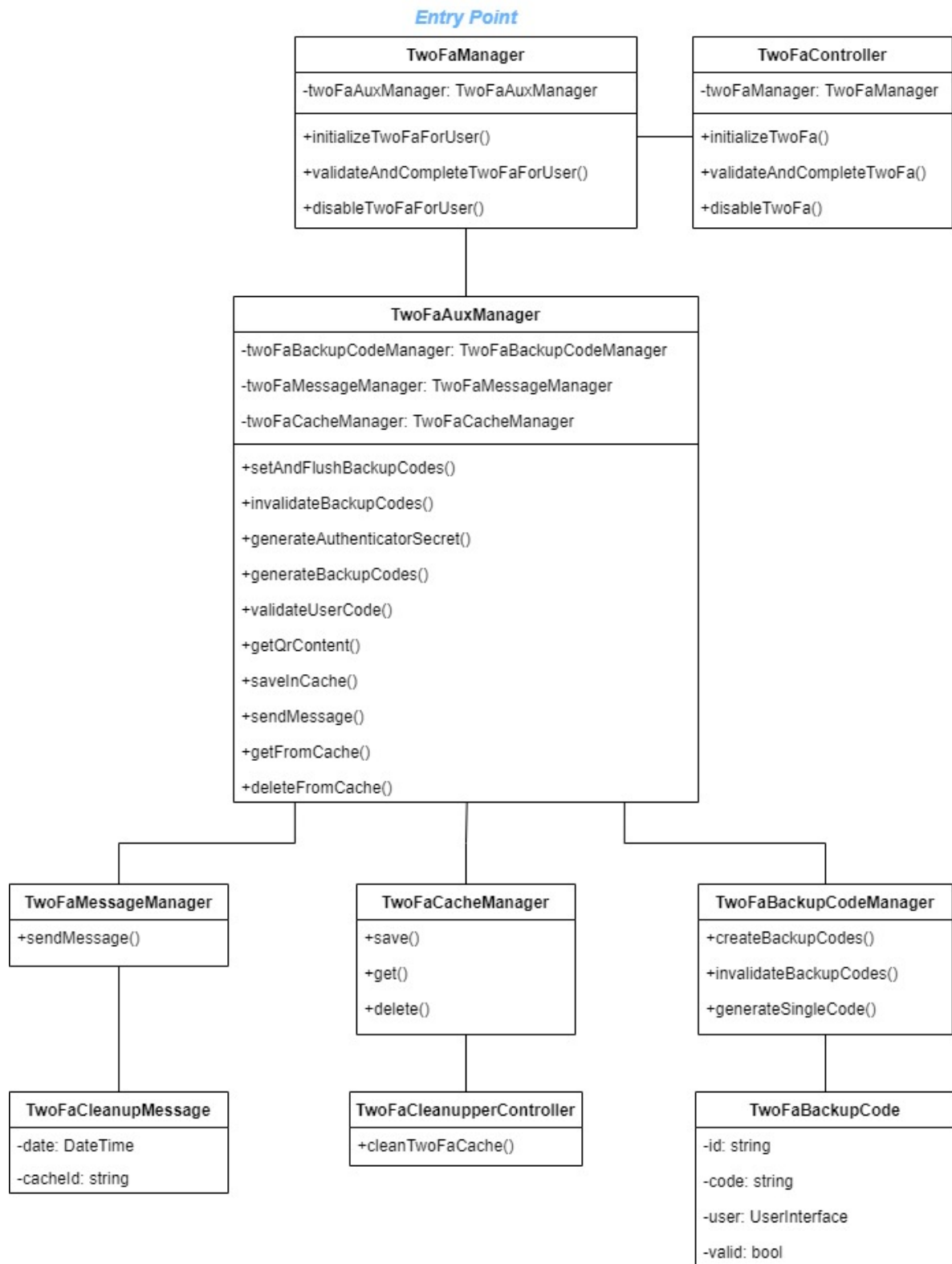


Figura 4.4: Diagramma delle classi (*class diagram*) raffigurante la struttura interna del servizio in fase di progettazione

5

Implementazione

In questo capitolo verrà discussa l'implementazione del servizio di autenticazione a due fattori facendo riferimento a tutte le linee guida e i concetti delineati precedentemente. La fase implementativa è suddivisa in tre stadi:

1. Implementazione del servizio da poter utilizzare tramite *web browser*
2. Sviluppo di una *REST API* affinché le applicazioni *mobile* possano usufruire del servizio
3. Creazione di un *bundle* che racchiuda le funzionalità del servizio in modo tale da poterne usufruire su qualsiasi piattaforma web

5.1 Tecnologie utilizzate

Il *backend* della piattaforma web da cui partire per sviluppare il servizio utilizza *Symfony* come *framework* principale. Si tratta di un *framework PHP (open-source)* per lo sviluppo di applicazioni web che consente di creare sistemi, anche complessi, in breve tempo coniugando solidità e velocità di sviluppo.

Symfony, inizialmente, era basato sull'architettura *MVC (Model-View-Controller)* ma, dalla versione 2, predilige l'approccio *Request-Response*. Il *framework*, inoltre, è compatibile con i principali gestori di basi di dati e consente l'indipendenza dal particolare *DBMS (Database management system)* adottato. Permette, peraltro, di automatizzare alcune attività dello sviluppo di progetti web (come la gestione della *cache* e l'autenticazione tramite credenziali di accesso) e consente di integrare diverse librerie *PHP* (componenti) in modo rapido e flessibile.

Symfony è composto da diversi componenti contraddistinti da un basso livello di accoppiamento. Due componenti, in particolare, saranno fondamentali durante l'implementazione del servizio: *Twig* e *Doctrine*. Il primo è un *template engine* per *PHP* progettato dagli stessi sviluppatori di *Symfony*. Nonostante *PHP* stesso possa essere usato come un *template engine*, negli anni non si è più evoluto in questa direzione e, pertanto, non supporta molte delle funzionalità offerte dai *template engine* moderni. Dovendo gestire numerose pagine, inoltre, la mancanza di un *template engine* causerebbe molti rallentamenti e renderebbe il processo molto macchinoso. *Twig* semplifica, quindi, la gestione dei *template* e offre i seguenti vantaggi:

- sintassi semplice ed efficace

- *shortcut* per gestire i casi d'uso più comuni
- *template* base estendibili (ereditarietà) e adattabili al contesto in maniera semplice (non è più necessario duplicare il codice)
- l'*overhead* rispetto a codice *PHP vanilla* è ridotto al minimo
- possibilità di estendere la libreria in caso non bastassero le funzionalità di base

Doctrine (*Doctrine Project*), invece, è una raccolta di librerie e strumenti *PHP open-source* che si focalizzano sulla gestione dell'archiviazione e della mappatura degli oggetti in *database*. I progetti principali sono *Object Relational Mapper (ORM)* e *Database Abstraction Layer (DBAL)*.

Doctrine permette di semplificare le operazioni con il *database* ed è possibile immaginarlo come un ponte che collega l'applicazione e i dati nel *database* (*middleware*). *ORM* definisce un particolare sistema che consente di interfacciarsi con il *database* gestendo i dati come se fossero veri e propri oggetti *PHP*. I dati sono memorizzati in tabelle e, quando viene effettuata una *query*, ogni riga estratta viene convertita (mappata) in automatico in un oggetto *PHP*.

Un'altra caratteristica di *Doctrine* è la capacità di scrivere *query* in un dialetto *SQL* orientato agli oggetti chiamato *DQL (Doctrine Query Language)*. La classe *QueryBuilder*, inoltre, permette di creare *query* attraverso un'interfaccia flessibile e intuitiva.

Per gestire le dipendenze, inoltre, verrà utilizzato *Composer*: un gestore di pacchetti a livello applicativo per *PHP*. *Composer* viene eseguito a linea di comando e fornisce un formato standard per la gestione delle dipendenze dei progetti *PHP* e delle librerie richieste.

La piattaforma web sui cui implementare il servizio utilizza i servizi storage di *Google Cloud* ma, durante la fase di implementazione, verrà creato un *database (PostgreSQL)* su una macchina virtuale per testare le nuove funzionalità in locale. Sulla macchina virtuale, inoltre, verrà installato anche *Apache HTTP Server*, un server web *HTTP open-source*. Quest'ultimo gestirà le richieste di risorse in arrivo (pagine, *file* eseguibili, *file* compressi, ecc.) utilizzando il protocollo *HTTP*. In questo modo si potrà accedere alla piattaforma web in locale.

5.1.1 Scelta del servizio di *Cloud Storage*

Nel precedente capitolo si è stabilito di utilizzare i servizi di *Cloud Storage* per simulare il comportamento di una *cache*. È necessario, quindi, selezionare una *suite* di servizi di *cloud computing*: dato che l'azienda ospitante (*Cogito srl*) possiede già piattaforme web che si basano su *Google Cloud Platform (GCP)*, si è preferito scegliere questa piattaforma.

Sempre nel capitolo riguardante la progettazione del servizio, inoltre, si è deciso di optare per un database di tipo *NoSQL*. I database *NoSQL* offerti da *GCP* includono: **Cloud Bigtable**, **Firestore**, **Firebase Realtime Database**, **Memorystore**.

Memorystore è il servizio ideale per implementare servizi di *cache*: utilizza il modello chiave-valore, assicura una bassa latenza e offre un'elevata scalabilità e disponibilità. Il servizio supporta sia *Redis* che *Memcached* (si tratta di *data store open-source* dedicati al *caching*). Il problema principale, però, è il costo: selezionando il piano base con una capacità di 1GB viene stimato un costo che si aggira tra i 30

e i 40 euro al mese. Vista la dimensione ridotta del servizio in fase di progettazione, si è deciso (almeno inizialmente) di optare per altri servizi.

Cloud Bigtable è un servizio che permette di salvare e gestire grandi quantità di dati in un archivio chiave-valore offrendo un'elevata scalabilità e una bassa latenza. Viene utilizzato nel mondo dei *Big Data* e non è adatto ad un servizio di *caching*.

Firestore Realtime Database è un *database* in cui i dati sono salvati sotto forma di documenti *JSON* (*JavaScript Object Notation*) e vengono sincronizzati in tempo reale su tutti i *client* connessi. Al posto di utilizzare richieste *HTTP* (*HyperText Transfer Protocol*), ogni volta che i dati salvati nel *database* subiscono delle modifiche ogni dispositivo riceve immediatamente la modifica. È una soluzione efficiente e a bassa latenza per applicazioni *mobile* collaborative e multi-utente che richiedono stati sincronizzati tra i *client* in tempo reale.

Firestore è l'evoluzione di *Firestore Realtime Database*. Fornisce molte delle funzionalità di quest'ultimo (sincronizzazione in tempo reale, modalità *offline*) offrendo allo stesso tempo un modello di dati più intuitivo, la possibilità di effettuare *query* anche complesse in maniera flessibile e veloce e, infine, la capacità di scalare in maniera ottimale. I dati non sono più organizzati in un semplice albero *JSON* ma in modo più strutturato e gerarchico: vengono salvati in documenti che sono a loro volta organizzati in collezioni. *Firestore*, inoltre, offre anche un piano base con il quale è possibile utilizzare il servizio gratuitamente. Il piano base garantisce, giornalmente, 1GB di dati da poter memorizzare e 50000 letture, scritture e cancellazioni. Se il piano base risulta troppo limitato, il costo del servizio verrà calcolato in base alla quantità di dati memorizzati nel *database*, al numero di documenti letti, scritti e cancellati e in base a quanta banda di rete è stata utilizzata.

Dopo aver analizzato i *database NoSQL* messi a disposizione da *GCP*, si è preferito optare per *Firestore* in quanto, visto il numero abbastanza ridotto di utenti delle piattaforme web dell'azienda, il piano base gratuito risulta più che sufficiente per la realizzazione del progetto. Nonostante questo servizio non sia principalmente incentrato sulla creazione di *database* per il *caching*, nella fasi iniziali del progetto risulta comunque una valida alternativa.

5.2 Implementazione del servizio per browser web

5.2.1 Installazione del bundle 2FA

Per installare il *bundle* selezionato nella fase di progettazione (*scheb/2fa*), sarà necessario aggiungerlo come dipendenza nel progetto della piattaforma web utilizzando *Composer*. Per facilitare l'installazione è possibile utilizzare *Symfony Flex*, un *plugin* di *Composer* che consente di eseguire delle azioni prima o dopo l'esecuzione dei comandi messi a disposizione da *Composer* (*require*, *update*, *remove*). A titolo di esempio, quando viene eseguito un *require* e *Symfony Flex* è installato correttamente, l'applicazione inoltra una richiesta al server di *Symfony Flex* prima di installare il *bundle* tramite *Composer*. Se non viene trovata nessuna corrispondenza per quello specifico *bundle*, quest'ultimo verrà installato tramite le normali procedure di *Composer*. Se, invece, viene trovata una corrispondenza, il server fornirà una "ricetta" (*recipe*) che verrà utilizzata dall'applicazione per decidere quale *bundle* installare e quali azioni dovranno essere svolte in automatico una volta conclusa l'installazione. *Symfony Flex*, quindi, permette di automatizzare tutte le azioni necessarie per installare correttamente un *bundle*.

L'installazione del bundle, pertanto, richiederà solamente l'esecuzione della seguente riga di comando se *Symfony Flex* è stato installato:

```
> composer require 2fa
```

In questo modo verrà installato il *bundle* “base”. È possibile, inoltre, installare dei pacchetti aggiuntivi per poter usufruire di alcune funzionalità che non sono state inserite all'interno del *bundle* “base”. Per lo sviluppo del servizio in questione, sarà necessario installare il pacchetto che consente l'utilizzo dei codici di backup e il pacchetto che, utilizzando una specifica implementazione dell'algoritmo *TOTP*, permetterà al servizio di essere compatibile con l'applicazione *Google Authenticator*:

```
> composer require scheb/2fa-backup-code
> composer require scheb/2fa-google-authenticator
```

Successivamente, sarà necessario definire un *file* di configurazione in cui specificare le nuove rotte (percorsi) da utilizzare durante l'autenticazione. *Symfony Flex*, subito dopo l'installazione del *bundle*, ha generato automaticamente il *file* `config/routes/scheb_2fa..`. Non resta, dunque, che modificare le rotte già definite all'interno di questo *file*: la prima definirà il percorso tramite il quale accedere al *form* che richiederà il codice *2FA* all'utente, mentre la seconda identificherà il percorso che consentirà di validare il codice inserito dall'utente. È importante, inoltre, che i percorsi specificati si trovino all'interno del *pattern* stabilito dal *firewall* designato all'utilizzo dell'autenticazione a due fattori. In questo caso il *firewall* è `admin` (gestisce le rotte dell'area *admin* della piattaforma web) e, pertanto, le due rotte sono state definite come segue: `/admin/2fa` e `/admin/2fa_check`.

Il prossimo passo consiste nel configurare correttamente il *firewall* in questione (`admin`) affinché sia possibile attivare il *2FA*. Nel *file* `config/packages/security.yaml`, all'interno della sezione riguardante il *firewall* `admin`, dovrà essere aggiunto il campo `two_factor`:

```
# config/packages/security.yaml
security:
    firewalls:
        admin:
            ...
            two_factor:
                auth_form_path: 2fa_login      # Il nome della rotta definita in routes.yaml
                check_path: 2fa_login_check   # Il nome della rotta definita in routes.yaml
            ...
```

Nello stesso *file*, inoltre, è necessario aggiungere due ulteriori elementi al campo `access_control`

```
# config/packages/security.yaml
...
access_control:
    - { path: ^/admin/logout$, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/admin/2fa, role: IS_AUTHENTICATED_2FA_IN_PROGRESS }
    ...
```

Il primo elemento rende la rotta che gestisce il *logout* accessibile durante l'autenticazione a due fattori (`IS_AUTHENTICATED_ANONYMOUSLY` identifica la situazione in cui un utente non ha ancora effettuato il *login*). In questo modo, l'utente ha la possibilità di abbandonare il processo di autenticazione in qualsiasi momento.

Il secondo elemento, invece, stabilisce che il *form* in cui l'utente deve inserire il codice temporaneo potrà essere accessibile solo quando l'autenticazione a due fattori è realmente in corso. Per gestire questa situazione, è stato creato un nuovo ruolo (`IS_AUTHENTICATED_2FA_IN_PROGRESS`) che identifica l'utente che ha effettuato il *login* ma che non ha ancora inserito un codice di autenticazione a due fattori valido.

Dopodiché, è necessario apportare alcune modifiche al file `config/packages/scheb_2fa.yaml`. Per prima cosa, nel campo `security_tokens` bisogna scegliere il metodo di *login* che il *firewall* della piattaforma web utilizza. Dato che quest'ultima si avvale di un *form* in cui bisogna inserire *username* e *password*, si dovrà selezionare il seguente *token* di sicurezza:

```
# config/packages/scheb_2fa.yaml
...
security_tokens:
    - Symfony\Component\Security\Core\Authentication\Token\UsernamePasswordToken
...
```

Sarà necessario, inoltre, procedere con l'attivazione vera e propria dei *bundle* installati precedentemente. Per quanto riguarda il *bundle* che permette l'utilizzo dei codici di backup, è necessario aggiungere le seguenti righe nel file precedente:

```
# config/packages/scheb_2fa.yaml
...
backup_codes:
    enabled: true
...
```

I codici di backup, peraltro, devono essere forniti dall'oggetto utente e, perciò, l'entità utente deve implementare l'interfaccia `Scheb\TwoFactorBundle\Model\BackupCodeInterface` fornita dal *bundle*. L'entità utente, dunque, dovrà contenere una nuova proprietà che rappresenti i codici di backup associati all'utente, un metodo per controllare se si tratta di un codice di backup valido, un metodo per invalidare un codice di backup e un metodo per associare un nuovo codice di backup all'utente.

Il *bundle* che consente la compatibilità del servizio con l'applicazione *Google Authenticator*, necessita dell'aggiunta delle seguenti righe (sempre in `config/packages/scheb_2fa.yaml`) per poter essere attivato:

```
# config/packages/scheb_2fa.yaml
...
google:
    enabled: true
    server_name: Server Name
    issuer: Issuer Name
    digits: 6
```

```

window: 1
template: # Template used to render the authentication form
...

```

In questo modo si specifica il nome del server e il nome dell'emittente del codice che dovranno essere presenti all'interno del codice *QR*. Vengono definiti, inoltre, il numero di cifre dei codici di autenticazione da creare, quanti codici prima e dopo l'attuale verranno accettati come validi (*delay window*) ed il template che verrà utilizzato per renderizzare il *form* di autenticazione (se non specificato verrà utilizzato un *template* predefinito). L'entità utente, inoltre, dovrà implementare l'interfaccia `Scheb\TwoFactorBundle\Model\Google\TwoFactorInterface`. Quest'ultima richiede la creazione di una nuova proprietà associata all'entità utente per rappresentare il codice segreto di autenticazione e l'aggiunta di tre ulteriori metodi: un metodo per verificare se l'autenticazione a due fattori è attiva, un metodo per ottenere lo *username* dell'utente e un metodo per ottenere il codice segreto associato all'utente.

5.2.2 Attivazione del 2FA

In questa sezione verrà discussa l'implementazione delle due fasi che consentiranno l'attivazione del 2FA per un determinato utente. Dato che, inoltre, la piattaforma web dispone di una pagina contenente una lista di tutti gli utenti registrati nel sistema, si è deciso di inserire un'azione (*action*) per ogni utente (`User`) presente nella lista. L'azione dovrà consentire l'attivazione, la disattivazione e la rigenerazione del 2FA a seconda dello stato in cui si trova l'utente.

Per far ciò è necessario introdurre due nuove rotte (`enable-2fa-step-1` e `disable-2fa`) nell'*admin* responsabile della precedente pagina (`UserAdmin`) e associare un *template* (sviluppato in *Twig*) all'azione precedentemente creata per definire i nuovi pulsanti (Attiva 2FA, Disattiva 2FA, Rigenera 2FA) e i loro collegamenti (ovvero verso quale indirizzo instradare l'utente).

Fatto ciò, sarà compito del *controller* di `UserAdmin` (`UserAdminController`) gestire il comportamento della piattaforma web quando l'utente viene indirizzato verso le due nuove rotte. Per questo motivo, dunque, dovranno essere creati due nuovi metodi nel *controller* per gestire l'attivazione e la disattivazione del 2FA: `enable2faStep1Action` e `disable2faAction`.

Prima fase

Il metodo `enable2faStep1Action` (*Listing 1*) dovrà inizializzare il processo di attivazione del 2FA per un particolare utente e mostrare a quest'ultimo una pagina contenente i codici di backup, il codice *QR* per permettere l'associazione con *Google Authenticator* e un *form* in cui l'utente potrà inserire il codice temporaneo di autenticazione. L'inserimento del codice è necessario per ottenere la conferma che l'utente abbia veramente intenzione di attivare il 2FA. Una volta verificata la validità del codice, è possibile procedere con la seconda fase.

Per procedere con l'inizializzazione, `enable2faStep1Action` dovrà far riferimento a `TwoFaManager`. Quest'ultimo mette disposizione il metodo `initializeTwoFaForUser` che riceve in input un oggetto utente che implementa `UserInterface` (l'interfaccia che l'entità utente deve implementare affinché il *bundle* funzioni correttamente). Facendo riferimento a `TwoFaAuxManager`, il metodo:

- ricava una collezione di codici di backup
- genera il codice di autenticazione segreto e lo associa all'oggetto utente in questione
- ottiene il contenuto del codice *QR* da mostrare all'utente
- crea un *DTO* (*Data Transfer Object*) in cui salvare l'oggetto utente, i codici di backup e il contenuto del codice *QR*
- salva il *DTO* in *cache*
- invia un messaggio a *Google Cloud Pub/Sub* contenente il nome del documento con cui è stato salvato il *DTO* in *cache*.

Il metodo `initializeTwoFaForUser`, infine, restituisce il *DTO* (`UserTwoFaConfiguration`) creato all'interno della procedura.

Un *DTO* (*Data Transfer Object*) è un *design pattern* usato per trasferire dati tra sottoinsiemi di un'applicazione software [15]. In questo caso si tratta di un oggetto utilizzato per incapsulare i dati che dovranno essere inseriti in *cache*, evitando transazioni multiple. Questa tipologia di oggetti, inoltre, non dovrebbe avere alcun comportamento se non quello di archiviare e recuperare i suoi dati.

Una volta ottenuto il *DTO* contenente i dati salvati in *cache*, sarà compito di `enable2faStep1Action` creare il *form* in cui l'utente potrà inserire il codice temporaneo. È opportuno, quindi, creare una classe (`TwoFaActivationValidatorType`) che gestisca il *form* e i suoi campi. Verrà definito:

- un campo (obbligatorio) per ricevere il codice inserito dall'utente controllando, al tempo stesso, che il codice consista di 6 cifre
- un campo nascosto all'utente (*hidden*) contenente il nome (*key*) con cui il *DTO* è stato salvato in *cache*
- un pulsante di *submit* per inviare i dati inseriti nel *form*

Per questioni di sicurezza, si è deciso di salvare il *DTO* in *cache* con una chiave (*key*) ottenuta dall'*hashing* (*SHA-512*) di una variazione del codice di autenticazione segreto associato all'utente. Il metodo responsabile di eseguire questa procedura verrà implementato all'interno del *DTO*. La destinazione dei dati (*action*), inoltre, sarà una nuova rotta (`enable-2fa-step-2`) dichiarata nell'*admin* dell'utente. Quest'ultima verrà gestita dal metodo `enable2faStep2Action` di `UserAdminController` che si occuperà della seconda fase della procedura di attivazione del *2FA*.

Definito il *form*, non rimane che renderizzare il *template* da associare alla pagina web relativa alla prima fase. Al *template* vengono forniti:

- i codici di backup
- il codice *QR*
- il nome con cui è stato salvato il *DTO* in *cache*
- il *form*

- il *link* che consente all'utente di tornare alla lista utenti

Si è deciso di progettare un *template* relativamente semplice ma che svolga in maniera efficace la sua funzione. Per quanto riguarda l'implementazione, sono stati utilizzati i costrutti di base dell'*HTML* e del *CSS*. Per velocizzare lo sviluppo, inoltre, è stato utilizzato *Bootstrap*, un *framework open-source* che contiene modelli di progettazione basati su *HTML* e *CSS* per le varie componenti dell'interfaccia di un'applicazione web. Si è ricorso, infine, a *Javascript* per consentire il *download* di un *file* di testo contenente i codici di backup dell'utente. Una volta renderizzato, la pagina web risultante è mostrata in Figura 5.1.

```
public function enable2faStep1Action(User $user): Response
{
    $userTwoFaConfiguration = $this->getTwoFaManager()->initializeTwoFaForUser($user);

    $form = $this->createForm(TwoFaActivationValidatorType::class, null, [
        'action' => $this->generateUrl('enable-2fa-step-2', [
            'id' => (string) $user->getId()
        ]),
        'hash' => $userTwoFaConfiguration->getHash(),
    ]);

    return $this->renderWithExtraParams('@CogitowebTwoFa/show__2fa_qr_code.html.twig', [
        'qrCode' => new QrCode($userTwoFaConfiguration->getQrCodeContent()),
        'backupCodes' => $userTwoFaConfiguration->getBackupCodeCollection(),
        'hash' => $userTwoFaConfiguration->getHash(),
        'link' => $this->admin->generateUrl('list', [
            'filter' => $this->admin->getFilterParameters()
        ]),
        'form' => $form->createView(),
        'errorMessage' => '',
    ]);
}
```

Listing 1: Metodo `enable2faStep1Action`

Dopo aver delineato il metodo `enable2faStep1Action`, è opportuno definire come sono stati implementati i componenti che hanno permesso a `initializeTwoFaForUser` di effettuare tutte le azioni correlate alla prima fase dell'attivazione del *2FA*.

Per quanto riguarda la generazione dei codici di backup, è necessario innanzitutto creare una classe (`TwoFaBackupCode`) che li rappresenti. Un codice di backup è caratterizzato da:

- un *id* che lo identifica univocamente
- un codice di backup vero e proprio (un numero compreso tra 100000 e 999999)
- l'utente a cui è associato
- un campo *booleano* che ne attesta la validità

Definita l'entità `TwoFaBackupCode`, è necessario creare un manager (`TwoFaBackupCodeManager`) che consenta di gestire i codici di backup. Il *bundle* utilizza un manager di default ma offre la possibilità di sostituirlo con uno personalizzato. Per far ciò sarà sufficiente creare un servizio (`TwoFaBackupCodeManager`)

Two-Factor Authentication


Add an extra layer of security to your account by requiring a numeric code when you log in.
You will need an app like **Google Authenticator** to enable it.

Below you can find 10 backup codes and a QR Code that will allow you to successfully authenticate in your account.
This extra security measure requires you to verify your identity using a 6-digit passcode (TOTP) generated by scanning the QR Code below with an authenticator.

Remember to save your backup codes and do not let anyone know them!

342761	854588	519751	784064	646584	592102	585020	624362	219612	502817
--------	--------	--------	--------	--------	--------	--------	--------	--------	--------

[Download them as text file](#)



Enter 6-digit code from your authenticator application*

[Back](#) [Activate 2FA](#)

Figura 5.1: Pagina web associata alla prima fase dell'attivazione del 2FA

che implementi l'interfaccia `BackupCodeManagerInterface` fornita dal *bundle* `scheb/2fa-backup-code` e, successivamente, registrarlo nelle configurazioni del *bundle*:

```
backup_codes:
  enabled: true
  manager: Cogitoweb\TwoFaBundle\Manager\TwoFaBackupCodeManager
```

`TwoFaBackupCodeManager` mette a disposizione i seguenti metodi:

- `createCodes` permette di generare 10 codici di backup (`TwoFaBackupCode`) da associare ad un utente specifico
- `invalidateBackupCodes` consente di invalidare tutti i codici di backup associati ad un utente
- `isBackupCode` controlla che il codice fornito in input sia valido per quell'utente
- `invalidateBackupCode` invalida il codice di backup passato in input e salva le modifiche effettuate in *database*

Una volta implementato, `TwoFaBackupCodeManager` verrà richiamato da `TwoFaAuxManager` per generare i codici di backup.

La generazione del segreto di autenticazione da associare all'utente viene delegata a `TwoFaAuxManager` che, a sua volta, richiama la classe `GoogleAuthenticator` del *bundle*. Quest'ultima permette di creare un segreto di 256 bit che verrà trasformato in una stringa di caratteri, codificata in *Base32*, per poter essere trasferito all'applicazione *Google Authenticator*. *Base32* è uno standard di codifica descritto in *RFC 4648* che facilita la trasmissione di stringhe binarie utilizzando un insieme di 32 cifre, ognuna delle quali può essere rappresentata da 5 bit. Una volta ottenuto il codice segreto, quest'ultimo viene assegnato all'utente che ha richiesto l'attivazione del *2FA*. La classe `GoogleAuthenticator`, inoltre, si occupa di generare anche il contenuto del codice *QR*. Quest'ultimo conterrà il codice segreto di autenticazione, il nome del server (opzionale) e il nome dell'emittente (opzionale).

Per quanto riguarda il salvataggio del *DTO* in *cache*, è necessario far riferimento a `TwoFaCacheManager`. Quest'ultimo, tramite il metodo `save` ricava il `FirestoreClient` e lo inizializza con le credenziali presenti all'interno della chiave privata fornita dall'account associato a *Google Cloud*. In questo modo si è effettuato il collegamento con il progetto *Google Cloud* associato alla piattaforma web ed è perciò possibile utilizzare tutte le funzionalità di *Firestore*. In seguito, il metodo crea una raccolta al cui interno viene salvato un documento che a sua volta conterrà il *DTO* serializzato. Il nome del documento viene ricavato dal metodo `getHash` (presente nel *DTO*) seguendo la procedura descritta in precedenza.

Il metodo, infine, si affida a `TwoFaAuxManager` per l'invio del messaggio a *Google Cloud Pub/Sub*. Ciò si rende necessario per gestire l'eliminazione automatizzata dei dati in *cache* (come spiegato nel capitolo precedente). Affinché *Google Cloud Pub/Sub* possa ricevere il messaggio, è necessario recarsi nella *console* della *Google Cloud Platform* e creare un nuovo *topic* nella sezione dedicata a *Pub/Sub*. In questo modo sarà possibile ricevere tutti i messaggi diretti a quel *topic* inviati da un certo *publisher*.

Un messaggio, all'interno del servizio, verrà rappresentato dall'entità (classe) `TwoFaCleanupMessage`. Quest'ultima contiene tre proprietà: la data e l'ora attuale (`DateTime`), il nome del documento in *cache* e il nome dell'evento che ha causato l'invio del messaggio.

Per gestire l'invio del messaggio, `TwoFaAuxManager` si affida al componente `TwoFaMessageManager`. Quest'ultimo mette a disposizione un metodo che permette di creare un nuovo messaggio di tipo `TwoFaCleanupMessage` e di inviarlo a *Google Cloud Pub/Sub*.

L'invio del messaggio vero e proprio è reso possibile dal componente *Messenger*, messo a disposizione da *Symfony* e installabile tramite *Composer* con il seguente comando:

```
> composer require symfony/messenger
```

Messenger fornisce un *message bus* che consente sia l'invio di messaggi da gestire immediatamente nella propria applicazione sia l'inoltro di messaggi tramite dei *transport* per poi gestirli in un secondo momento. Per comunicare con *Pub/Sub*, verranno sfruttati i *transport* in quanto consentono di inviare messaggi ad un sistema di *queueing* (es. *Pub/Sub*). *Messenger*, tuttavia, non supporta alcun *transport* che consente di inviare messaggi a *Google Pub/Sub*. Per far ciò è necessario installare i seguenti pacchetti:

```
> composer require sroze/messenger-enqueue-transport
```

```
> composer require enqueue/gps
```

```
> composer require enqueue/enqueue-bundle
```

Per prima cosa bisogna modificare il *file* contenente le impostazioni di *Messenger*: è necessario definire un *transport* tramite il quale i messaggi saranno inviati. Per far ciò, nel *file* `config/packages/enqueue.yaml` (*file* di configurazione di *EnqueueBundle*) verrà definito un *transport* da utilizzare di default. Dovendo comunicare con *Pub/Sub*, si dovrà definire:

- il *DSN* (*Database Source Name*)
- l'*id* del progetto *Google Cloud* a cui ci si vuole connettere
- il *file* contenente le credenziali dell'account *Google Cloud* del progetto
- la *factory* che si occuperà di inizializzare la connessione

```
# config/packages/enqueue.yaml
enqueue:
    default:
        transport:
            dsn: '%env(resolve:ENQUEUE_DSN)%'
            projectId: '%env(resolve:GOOGLE_CLOUD_PROJECT_ID)%'
            keyFile: '%env(resolve:GOOGLE_CLOUD_KEY)%'
            connection_factory_class: App\Enqueue\GpsConnectionFactory
            client: null
```

Nella precedente configurazione sono state richiamate alcune variabili d'ambiente contenute nel *file* `.env.local` della piattaforma web.

Definito il *transport* tramite il *bundle* *Enqueue*, quest'ultimo verrà elencato tra quelli a disposizione per il *bundle* *Messenger* nel *file* `config/packages/messenger.yaml`. All'interno dello stesso *file*, inoltre, dovrà essere indicata la classe utilizzata per la serializzazione (e deserializzazione) dei dati. Per quanto riguarda il servizio *2FA*, viene utilizzato il *JMSSerializer* che offre diversi strumenti per la conversione

degli oggetti del sistema in stringhe. Infine, per ogni tipo di messaggio (identificato dal proprio *FQCN*¹) viene associato il *transport* a cui trasmetterlo.

```
# config/packages/messenger.yaml
framework:
    messenger:
        serializer:
            default_serializer: App\Messenger\Serializer\JsonSerializer
        transports:
            gps: enqueue://default

        routing:
            'Cogitoweb\TwoFaBundle\Message\Cleanup2faMessage': gps
```

Quando *TwoFaMessageManager* richiamerà il metodo *dispatch* della classe *MessageBus* offerta da *Messenger*, il messaggio da inviare e i relativi metadati verranno inseriti in un contenitore (*Envelope*) che verrà successivamente serializzato. Solo dopo aver fatto ciò, il messaggio sarà gestito dal *transport* corrispondente e inviato a *Google Pub/Sub*.

Affinché *Google Pub/Sub* possa ricevere il messaggio, è necessario creare un *topic* ad-hoc (tramite la *console* della *Google Cloud Platform*) con lo stesso nome utilizzato all'interno della configurazione di trasporto del messaggio. Il *topic*, una volta ricevuto il messaggio, lo invierà a tutti i *subscriber* iscritti a quello specifico *topic*. È necessario, pertanto, definire un nuovo *subscriber* e creare una *subscription* al precedente *topic*.

L'obiettivo è trasmettere il messaggio alla piattaforma web in modo tale da gestire l'eliminazione automatizzata dei dati in *cache*. La nuova *subscription*, quindi, dovrà inviare i messaggi ad un determinato *endpoint* dell'applicazione web (impostare *push* come tipo di consegna). Come anticipato nel capitolo precedente, inoltre, è necessario impostare una *retry policy* che imponga un *backoff* di 10 minuti prima di rispedire un messaggio per il quale non è stato ricevuto un *acknowledgment* entro 30 secondi dall'invio oppure nel caso in cui venga ricevuto un *NACK* (*Negative Acknowledge Character*). In questo modo si evita di sovraccaricare la piattaforma web di messaggi provenienti da *Pub/Sub*.

Per quanto riguarda l'indirizzo (*endpoint*) a cui inviare il messaggio, è necessario definire una nuova rotta (*two_fa_cache_cleanupper*) che gestisca il nuovo percorso (*/2fa-cache-cleanupper*). Quest'ultimo, inoltre, per motivi di sicurezza dovrà contenere come parametro il *token* di autenticazione definito dal progetto. La rotta risultante verrà gestita dal *controller* *TwoFaCleanupperController*, il quale dovrà:

- recuperare il *token* dalla richiesta e controllare se si tratta di un *token* valido
- decodificare la richiesta, ricavando il contenuto del messaggio inviato da *Google Pub/Sub*
- ricavare il nome dell'evento associato al messaggio. Se l'evento è *2fa-cleaning*, allora:
 - dovrà ricavare il *timestamp* associato all'invio del messaggio e il nome del documento in *cache* in cui è stato salvato il *DTO*

¹Fully Qualified Class Name

- se sono trascorsi più di 10 minuti dal *timestamp* ottenuto al punto precedente, allora il documento in *cache* contenete il *DTO* verrà eliminato. Per indicare che l'elaborazione del messaggio è avvenuta con successo, verrà inviata una risposta con codice di stato 204 (*No content*). In caso contrario, invece, verrà trasmessa una risposta con codice di stato 409 (*Conflict*).
- Se l'evento non è *2fa-cleaning*, allora il messaggio non sarà processato ma verrà comunque inviata una risposta con codice di stato 204 (*No content*) per indicare la corretta ricezione ed elaborazione del messaggio a *Google Pub/Sub*.

Sorge però un problema: come testare il corretto funzionamento dell'eliminazione automatizzata dei dati in *cache* descritto precedentemente? *Google Cloud* necessita di un *endpoint* pubblico realmente esistente a cui trasmettere i messaggi (le richieste). In fase di sviluppo, però, si è utilizzato un server web locale (installato su una macchina virtuale) per eseguire la piattaforma web.

Per questo motivo si è deciso di ricorrere a *ngrok*, un *software* che consente di esporre un server web (eseguito in locale) su internet specificando su quale porta il server web è in ascolto e quale *host* utilizzare (tramite il campo `--host-header`). Una volta avviato, verrà visualizzata un'interfaccia utente direttamente nel terminale contenete l'*URL* pubblico del tunnel creato. L'ultimo passo, infine, consiste nell'inserire il precedente *URL* all'interno della *subscription* di *Google Pub/Sub* in modo tale da inviare (*push*) i messaggi a quel determinato *endpoint*.

Seconda fase

Al termine della prima fase di attivazione dell'autenticazione a due fattori è stata presentata all'utente una pagina contenete i codici di backup, il codice *QR* e un *form* in cui poter inserire il codice temporaneo ottenuto da *Google Authenticator*. I dati del *form* vengono inviati all'indirizzo definito dalla rotta `enable-2fa-step-2` (specificata nell'*admin UserAdmin*). Quest'ultima verrà gestita dal metodo `enable2faStep2Action` (*Listing 2*) del *controller UserAdminController*, responsabile della seconda fase del processo di attivazione.

Per prima cosa il metodo controlla la validità del *form* ricevuto: se quest'ultimo non è stato inviato tramite una richiesta *POST* oppure se ha superato la validazione lato *client* ma lato *server* sono stati individuati alcuni errori (il codice non è stato inserito, il codice non è di 6 cifre oppure il campo *hash* nascosto è stato modificato) significa che l'utente ha manomesso il *form*. In questo caso verrà visualizzato un messaggio d'errore e l'utente verrà reindirizzato alla lista utenti.

D'altro canto, se il *form* non contiene errori ed è stato inviato correttamente, vengono prelevati i dati contenuti al suo interno e richiamato il metodo `validateAndCompleteTwoFaForUser` (*Listing 3*) messo a disposizione da *TwoFaManager* per ultimare l'attivazione del *2FA*. Quest'ultimo dovrà:

- recuperare il *DTO* salvato in *cache* durante la prima fase (utilizzando come nome del documento il valore contenuto nel campo `hash` del *form*)
- assegnare all'utente il codice segreto di autenticazione fornito dal *DTO*
- validare il codice temporaneo inserito dall'utente

```

public function enable2faStep2Action(User $user, Request $request): Response
{
    $form = $this->createForm(TwoFaActivationValidatorType::class);
    $form->handleRequest($request);

    if (
        !$form->isSubmitted() or
        (
            !$form->isValid() and
            // If the user hacked the form, redirect back to the users list
            $form->get('hash')->getErrors()->count() > 0
        )
    ) {
        $this->addFlash(
            'sonata_flash_error',
            $this->trans('flash_2fa_invalid_hash_provided')
        );
        return $this->redirectToRoute('admin_app_user_list');
    }

    $code = $form->get('code')->getData();
    $hash = $form->get('hash')->getData();

    try {
        $this->getTwoFaManager()->validateAndCompleteTwoFaForUser($hash, $code, $user);
    } catch (UserTwoFaConfigurationNotFound $exception) {
        $this->addFlash(
            'sonata_flash_error',
            $this->trans('flash_2fa_invalid_hash_provided')
        );
        return $this->redirectToRoute('admin_app_user_list');
    } catch (InvalidTwoFactorCodeException $exception) {
        $errorMessage = $this->trans('flash_2fa_invalid_code_provided');

        return $this->renderWithExtraParams('@CogitowebTwoFa/show__2fa_qr_code.html.twig', [
            'qrCode' => new QrCode(
                $exception->getUserTwoFaConfiguration()->getQrCodeContent()
            ),
            'backupCodes' => $exception
                ->getUserTwoFaConfiguration()
                ->getBackupCodeCollection(),
            'hash' => $hash,
            'link' => $this->admin->generateUrl('list', [
                'filter' => $this->admin->getFilterParameters()
            ]),
            'form' => $form->createView(),
            'errorMessage' => $errorMessage,
        ]);
    }

    $this->addFlash(
        'sonata_flash_success',
        $this->trans('flash_2fa_enabled_successfully')
    );

    return $this->redirectToRoute('admin_app_user_list');
}

```

Listing 2: Metodo enable2faStep2Action

```

public function validateAndCompleteTwoFaForUser(
    string $hash,
    string $code,
    UserInterface $user
): void
{
    $userTwoFaConfiguration = $this->getTwoFaAuxManager()->get($hash);
    $user->setGoogleAuthenticatorSecret($userTwoFaConfiguration->getAuthenticatorSecret());

    if (!$this->getTwoFaAuxManager()->validateUserCode($user, $code)) {
        throw new InvalidTwoFactorCodeException($userTwoFaConfiguration);
    }

    $this->getTwoFaAuxManager()->deleteItem($hash);
    $this
        ->getTwoFaAuxManager()
        ->setAndFlushBackupCodes($user, $userTwoFaConfiguration->getBackupCodeCollection());
}

```

Listing 3: Metodo `validateAndCompleteTwoFaForUser`

Se il codice è corretto bisognerà:

- eliminare il documento in *cache* contenente il *DTO*
- invalidare i codici di backup assegnati precedentemente all'utente
- assegnare all'utente i nuovi codici di backup
- salvare i nuovi codici di backup e il codice segreto di autenticazione nel *database*

Se, al contrario, il codice non risulta valido, sarà necessario renderizzare nuovamente il *form* includendo, questa volta, un messaggio di errore. Nel caso in cui, invece, non esiste alcun documento in *cache* con quel nome oppure il documento non contiene alcun dato, l'utente o una persona esterna ha manipolato e compromesso il *form* creato dalla prima fase. Per questioni di sicurezza, quindi, la procedura di attivazione non deve essere portata a termine: l'utente verrà reinderizzato alla lista utenti con un messaggio d'errore opportuno.

Infine, se al termine di tutte queste operazioni non verrà sollevata alcuna eccezione, l'utente verrà riportato alla lista utenti con un messaggio che conferma che l'attivazione dell'autenticazione a due fattori è avvenuta con successo.

Una volta definito il comportamento del metodo `validateAndCompleteTwoFaForUser`, è opportuno delineare come sono stati implementati i componenti che hanno consentito di effettuare tutte le operazioni descritte precedentemente.

Per quanto riguarda il recupero del *DTO* salvato in *cache*, si dovrà far riferimento a `TwoFaAuxManager` che, a sua volta, richiamerà il metodo `get` del componente `TwoFaCacheManager`. Quest'ultimo riceve in input il nome del documento in *cache* contenente il *DTO*. Se il documento cercato non esiste o non contiene alcun dato, viene sollevata un'eccezione che verrà gestita dal metodo `enable2faStep2Action` seguendo la logica descritta precedentemente. D'altro canto, se il documento viene trovato con successo e non è vuoto, il *DTO* al suo interno verrà ricostruito (deserializzato) e restituito al chiamante.

Per effettuare la validazione del codice temporaneo inserito dall'utente, il metodo fa riferimento a `TwoFaAuxManager` che, a sua volta, richiama il metodo `checkCode` della classe `GoogleAuthenticator` fornita dal *bundle*.

L'eliminazione del documento in *cache*, invece, viene realizzata dal metodo `deleteItem` del componente `TwoFaCacheManager`. Il metodo riceve in input il nome del documento da eliminare e procede semplicemente all'eliminazione.

Per quanto riguarda l'invalidazione dei codici di backup, `TwoFaAuxManager` fornisce il metodo `setAndFlushBackupCodes`. Quest'ultimo richiama il metodo `invalidateBackupCodes` del componente `TwoFaBackupCodeManager` per invalidare i codici di backup precedentemente associati all'utente (ponendo a `false` il campo `valid` di ogni codice). Il metodo `setAndFlushBackupCodes`, infine, salva in *database* i nuovi codici di backup, il codice segreto di autenticazione e tutte le modifiche effettuate in precedenza.

5.2.3 Disattivazione del 2FA

Nella sezione 5.2.2 (Attivazione del 2FA) del presente capitolo, è già stato definito un pulsante (tramite un *template Twig*) che guiderà l'utente all'indirizzo definito dalla rotta `disable-2fa`. Tuttavia, per fare in modo che l'utente possa disattivare realmente il 2FA, è necessario definire il metodo `disable2faAction` (*Listing 4*) all'interno dello `UserAdminController` affinché sia possibile gestire la precedente rotta. Quest'ultimo farà riferimento a `disableFullTwoFaForUser` (*Listing 5*), fornito dal componente `TwoFaManager`, per:

- invalidare i codici di backup associati precedentemente all'utente
- eliminare (impostare a `null`) il codice segreto di autenticazione abbinato all'utente
- salvare in *database* le modifiche effettuate

L'invalidazione dei codici di backup verrà gestita dal metodo `invalidateBackupCodes` del componente `TwoFaBackupCodeManager`, già implementato nella fase di attivazione del 2FA.

Una volta che le precedenti operazioni si sono concluse con successo, l'utente verrà riportato alla pagina contenente la lista degli utenti del sistema e verrà mostrato un messaggio che conferma l'avvenuta disattivazione del 2FA.

5.2.4 Autenticazione tramite 2FA

L'autenticazione è quasi completamente gestita dal *bundle*. Quest'ultimo si collega al livello di sicurezza dell'applicazione (*security layer*) e, tramite dei *listener*, si mette in ascolto degli eventi che vengono scaturiti durante la fase di autenticazione.

Una volta che l'utente ha completato con successo l'autenticazione tramite credenziali, i metodi del *bundle* controllano se l'entità utente in questione possiede un codice di autenticazione segreto. In caso di risposta affermativa, verrà presentato un *form* in cui poter inserire il codice di autenticazione temporaneo. In questo momento, quindi, l'accesso viene negato e i privilegi (da utente autenticato) vengono temporaneamente trattenuti.

Il processo di autenticazione, a questo punto, si trova in uno stato intermedio (Figura 5.2). Per completare l'autenticazione, l'utente dovrà inserire un codice temporaneo fornito da *Google Authenticator*.

```

public function disable2faAction(User $user): RedirectResponse
{
    $this->getTwoFaManager()->disableFull2FaForUser($user);
    $this->addFlash(
        'sonata_flash_success',
        $this->trans('flash_2fa_disabled_successfully')
    );

    return new RedirectResponse(
        $this->admin->generateUrl('list', [
            'filter' => $this->admin->getFilterParameters()
        ])
    );
}

```

Listing 4: Metodo disable2faAction

```

public function disableFull2FaForUser(UserInterface $user): void
{
    $this->getTwoFaAuxManager()->invalidateBackupCodes($user);
    $user->setGoogleAuthenticatorSecret(null);

    $this
        ->getManagerRegistry()
        ->getManagerForClass(get_class($user))
        ->flush()
    ;
}

```

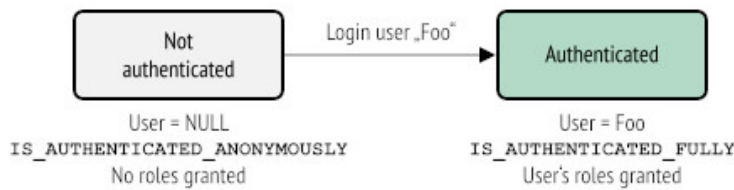
Listing 5: Metodo disableFull2FaForUser

Solo quando il codice verrà validato con successo l'utente potrà accedere al suo account e usufruire di tutti i privilegi associati.

Gli unici passaggi da seguire per consentire il corretto funzionamento dell'autenticazione a due fattori sono stati illustrati in dettaglio nella sezione 5.2.1 (Installazione del bundle *2FA*). Di seguito vengono riportate, in maniera sintetica, le azioni principali da compiere:

- definire una rotta tramite la quale è possibile visualizzare il *form* dell'autenticazione a due fattori
- definire una rotta che si occuperà di validare il codice di autenticazione temporaneo inserito dall'utente
- associare le nuove rotte all'*admin* in cui si vuole attivare l'autenticazione a due fattori
- specificare, nella configurazione del *bundle*, quale implementazione dell'algoritmo *TOTP* utilizzare e impostarne i parametri (nel caso del progetto è stata utilizzata l'implementazione di *Google Authenticator*)
- fare in modo che l'entità utente implementi l'interfaccia *TwoFactorInterface* fornita dal *bundle*

Standard Authentication



With Two-Factor Authentication

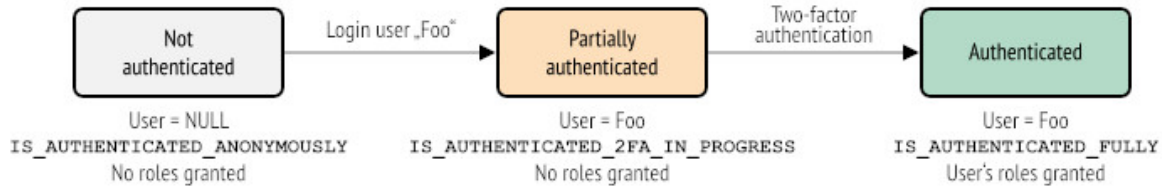


Figura 5.2: Autenticazione standard vs Autenticazione con 2FA (da [13])

5.3 REST API

Come già evidenziato nel capitolo precedente, affinché il servizio possa essere utilizzato anche da applicazioni *mobile*, è necessario sviluppare una *Web API* che metta a disposizione tutte le funzionalità offerte dal servizio. In questo modo, un'applicazione *client* esterna può usufruire dei servizi di una piattaforma web. Per far ciò, sarà necessario progettare un'interfaccia (accessibile dalle applicazioni *client*) che consenta di interagire con la logica del servizio offerto dalla piattaforma ma che, allo stesso tempo, ne nasconda l'implementazione.

L'*API* da progettare, inoltre, seguirà l'approccio *REST* ma dovrà essere di tipo *stateful*. È necessario, infatti, l'utilizzo di una sessione per memorizzare lo stato del *login*, indispensabile per poter verificare se l'utente ha già completato l'autenticazione a due fattori. La sessione verrà inizializzata dall'*API* e dovrà essere passata dal *client* ad ogni chiamata. Nelle configurazioni del *firewall* che gestirà il *login* tramite *API* bisognerà, pertanto, impostare il campo `stateless` a `false`.

Si è deciso, inoltre, di utilizzare *JSON* come formato dei messaggi da trasferire sia per la semplicità e compattezza rispetto all'*XML* sia perché la piattaforma web utilizza *json_login* come meccanismo di autenticazione tramite *API*. Quest'ultimo, quando viene inviata una richiesta di tipo *POST* alla rotta designata al *login*, cercherà automaticamente il contenuto *JSON* della richiesta, lo decodificherà, e utilizzerà i campi `username` e `password` per autenticare l'utente.

Affinché l'*API* possa funzionare, è necessario implementare le seguenti tre classi:

1. un *custom success handler* per il meccanismo di autenticazione
2. un *custom success handler* per il 2FA
3. un *custom failure handler* per il 2FA

Senza il 2FA, quando l'utente inserisce le credenziali, il sistema restituirà una delle seguenti risposte: “*login* eseguito con successo” oppure “*login* fallito”. D'altro canto, se il 2FA è attivo deve essere possibile inviare un terzo tipo di risposta: “autenticazione a due fattori richiesta”. Ricevendo questa

risposta, il *client* dovrà mostrare il *form* di autenticazione a due fattori all'utente. Per far ciò, è necessario progettare un *custom success handler*: la piattaforma web già presenta un *success handler* (`AuthenticationSuccessHandler`) che gestisce il *login* tramite *API*. Sarà necessario, quindi, effettuare soltanto delle piccole modifiche per gestire il caso in cui il *2FA* sia attivo per l'utente in questione.

Se si presenta questa situazione, l'*handler* dovrà inviare una risposta con codice di stato 401 (*Unauthorized*). Il codice indica che la richiesta del *client* non è stata completata perché non dispone di credenziali di autenticazione valide per la risorsa richiesta (*RFC 7235 [5]*). Il server, inoltre, deve inserire all'interno della risposta anche un campo di intestazione (`WWW-Authenticate`) che indichi quali tipologie di autenticazione possono essere usate per accedere alla risorsa e quali dati bisogna fornire per ogni schema di autenticazione. L'*handler*, quindi, invierà una risposta con il seguente campo di intestazione:

```
WWW-Authenticate: Totp realm="Two factor authentication required"
```

Totp (*Time-based One-Time Password*) rappresenta il metodo di autenticazione mentre **realm** contiene una breve descrizione dell'area protetta. Se il *2FA* non è attivo per l'utente in questione, l'*handler* dovrà semplicemente ritornare la risposta di *default* restituita nel caso in cui il *login* venga effettuato con successo.

È richiesta, inoltre, una risposta quando l'autenticazione a due fattori è stata completata con successo e l'utente ha completato l'intero processo di autenticazione. È necessaria, dunque, la creazione di un *handler* (`TwoFactorAuthenticationSuccessHandler`) che gestisca questa situazione. Per semplicità, si è scelto di non definire un nuovo tipo di risposta per questo tipo di evento ma di ritornare la risposta di *default* progettata nel caso in cui l'utente effettua il *login* con successo.

È opportuno, infine, definire una risposta anche quando è stato eseguito un tentativo di autenticazione tramite *2FA* ma, per qualche motivo, non è andato a buon fine. L'*handler* progettato (`TwoFactorAuthenticationFailureHandler`), infatti, dovrà strutturare la risposta in modo tale da comunicare al *client* che l'autenticazione tramite *2FA* non è stata eseguita con successo. L'*handler*, pertanto, invierà una risposta con codice di stato 400 (*Bad Request*), segnalando che il server non può processare la richiesta a causa di un errore del *client* (es. formato della richiesta non valido).

Gli *handler*, inoltre, dovranno essere registrati come servizi e richiamati negli appositi campi all'interno del *file* contenente la configurazione del *firewall* (`login`) che gestirà l'intera *API*. Una volta fatto ciò, è necessario specificare quale sarà la rotta (`2fa_api_login_check`) a cui inviare il codice inserito dall'utente per eseguirne la validazione. Prima di far ciò, però, la rotta deve essere dichiarata in `config/routes/scheb_2fa.yaml`, un *file* contenente tutte le rotte definite dal *bundle*.

In base alla documentazione del *bundle* e alle precedenti considerazioni, la configurazione del *firewall* dovrà essere la seguente:

```
# config/packages/security.yaml
login:
  pattern: ^/api/login
  stateless: false
  anonymous: true
  json_login:
```

```

    check_path: api_login
    success_handler: App\Security\Http\Authentication\AuthenticationSuccessHandler
    failure_handler: App\Security\Http\Authentication\AuthenticationFailureHandler
two_factor:
    prepare_on_login: true
    prepare_on_access_denied: true

    check_path: 2fa_api_login_check
    post_only: false

    success_handler:
        App\Security\Http\Authentication\TwoFactorAuthenticationSuccessHandler
    failure_handler:
        App\Security\Http\Authentication\TwoFactorAuthenticationFailureHandler

```

5.3.1 Definizione dell'API

Prima di addentrarsi nell'implementazione, è consigliato definire il comportamento che ci si aspetta dall'*API*, ovvero

- quali *endpoint* dovrà mettere disposizione
- quali operazioni verranno eseguite da ciascun *endpoint*
- per ogni operazione, quali sono gli *input* e gli *output* previsti

Così facendo, si definisce una documentazione che precisa come utilizzare l'*API*. Questo approccio, però, è caratterizzato dai seguenti problemi:

- documentazione poco chiara che può portare a diverse interpretazioni
- documentazione non esistente o incompleta
- informazioni obsolete
- informazioni scritte in un linguaggio che il lettore non conosce

Per questi motivi, si è scelto di adottare la specifica *OpenAPI*: un'iniziativa *open source* che definisce un formato per creare *file* di interfaccia leggibili dai computer per descrivere, produrre, consumare e visualizzare *API RESTful* e servizi web. Avendo a disposizione un *file* leggibile sia dagli umani che dai computer, è possibile automatizzare molti dei processi legati allo sviluppo di un'*API*. Partendo da questo *file*, infatti, è possibile generare un documento (leggibile dagli esseri umani) che fornisce una descrizione dettagliata dell'*API*.

Sono presenti, inoltre, numerosi strumenti che consentono di generare del codice *boilerplate* per utilizzare e implementare l'*API*. Ciò consente di focalizzarsi sulla *business logic* dell'*API* e di evitare discrepanze tra il codice e la documentazione.

Il *file* di interfaccia viene scritto in *YAML* o *JSON* in quanto definiscono formati facili da leggere e da gestire sia per i computer che per gli umani. Utilizzando, quindi, una specifica standardizzata, ogni utente che conosce le basi delle *API RESTful* può comprendere e interagire con il servizio offerto.

Per quanto riguarda il progetto, il *file* contenete la specifica dell'*API* offerta dall'applicazione web è `swagger.yaml`. All'interno di quest'ultimo, per consentire l'attivazione e la disattivazione del *2FA*, verranno aggiunti i seguenti *endpoint*:

- (POST) `/2fa/enable2fa`
- (POST) `/2fa/codeValidation`
- (POST) `/2fa/disable2fa`

Per permettere, invece, l'autenticazione tramite *2FA* verrà aggiunto il seguente *endpoint*:

- (POST) `/login2fa`

L'*endpoint* `enable2fa` esegue la prima fase associata all'attivazione del *2FA*: genera i codici di backup ed elabora il contenuto del codice *QR* da inviare via *e-mail* all'utente. Nel caso in cui le operazioni vadano a buon fine, l'*endpoint* ritorna un oggetto *JSON* contenente i codici di backup, il contenuto del codice *QR* e il nome con cui il *DTO* è stato salvato in *cache* (*hash*). Se l'utente, invece, cerca di attivare il *2FA* ma non si è autenticato, il server dovrà inviare una risposta con codice di stato 401 (*Unauthorized*).

Il *file* `swagger.yaml` permette di descrivere il comportamento del precedente *endpoint* in questo modo:

```
/2fa/enable2fa:
  post:
    security:
      - bearerAuth: [ ]
    description: |
      This endpoint should be used to generate the backup codes and the totp
    tags:
      - TwoFactorAuthentication
    responses:
      '200':
        description: Successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/TwoFaInfo'
      '401':
        $ref: '#/components/responses/Unauthorized'
      '500':
        $ref: '#/components/responses/InternalServerError'
```

A partire da questo *file*, inoltre, può essere generata una documentazione più facilmente consultabile dagli esseri umani. Per brevità, inoltre, verrà riportata solo la porzione del *file* associato al precedente *endpoint* (`enable2fa`): i restanti *endpoint* utilizzeranno la medesima struttura e sintassi.

L'endpoint `codeValidation` consente l'esecuzione della seconda fase dell'attivazione del *2FA*. Riceve in *input* un oggetto *JSON* contenente il codice inserito dall'utente e il campo `hash` (nome del documento contenente il *DTO* in *cache*) del *form* presentato all'utente. Nel caso in cui la richiesta venga processata correttamente, il server dovrà ritornare una risposta con codice di stato 204 (*No Content*). Questo tipo di risposta non contiene alcun dato ma comunica al *client* che la richiesta è stata elaborata con successo. Tuttavia, se la richiesta contiene un codice non valido o il campo *hash* fornito è errato, il *server* invierà una risposta con codice di stato 400 (*Bad Request*). Qualora l'utente, invece, stia cercando di accedere all'endpoint in questione senza aver eseguito l'autenticazione, il *server* trasmetterà una risposta con codice di stato 401 (*Unauthorized*).

Per quanto riguarda la disattivazione del *2FA*, viene fornito l'endpoint `disable2fa`: quest'ultimo, come il precedente *endpoint*, ritorna una risposta con codice di stato 204 (*No Content*) quando la richiesta è stata processata con successo. Se l'utente tenta di accedere all'endpoint in questione senza essersi autenticato, il *server* trasmetterà una risposta con codice di stato 401 (*Unauthorized*).

La validazione del codice temporaneo inserito dall'utente durante la fase di *login* viene effettuata dall'endpoint `login2fa`. Quest'ultimo riceve in *input* un oggetto *JSON* contenente lo *username*, la *password* e il codice temporaneo di autenticazione. Se il codice viene considerato valido, il *server* invia una risposta con codice di stato 200 (*OK*). Nel caso in cui, invece, il codice non sia valido, verrà restituita una risposta con codice di stato 400 (*Bad Request*). Infine, qualora l'utente stia cercando di accedere all'endpoint in questione senza prima essersi autenticato tramite *username* e *password*, il *server* dovrà inviare una risposta con codice di stato 401 (*Unauthorized*).

5.3.2 Implementazione dell'API

Per implementare i precedenti *endpoint*, sono state dichiarate le rotte associate ai percorsi:

- `/2fa/enable2fa`
- `/2fa/codeValidation`
- `/2fa/disable2fa`

La gestione delle nuove rotte viene affidata al *controller* `TwoFaController`. Quest'ultimo definirà la *business logic* da associare ad ogni *endpoint*.

Per quanto riguarda l'endpoint `enable2fa`, `TwoFaController` definisce il metodo `postEnable2faAction` (*Listing 6*) che svolge le seguenti operazioni:

- ricava l'utente (autenticato) che ha avviato la procedura di attivazione del *2FA*
- esegue la prima fase dell'attivazione del *2FA* utilizzando l'utente attuale come parametro (ottenendo il *DTO* salvato in *cache*)
- ricava il contenuto del codice *QR* e ne ottiene l'immagine
- invia un'e-mail all'utente in questione contenente i codici di backup, il codice *QR* e le istruzioni per inizializzare *Google Authenticator*.
- restituisce in output il *DTO* generato precedentemente

La prima fase dell'attivazione del *2FA* viene eseguita richiamando il metodo `initializeTwoFaForUser` del componente `TwoFaManager`. L'invio dell'*e-mail*, invece, è reso possibile da `SwiftMailerBundle`, un *bundle* fornito direttamente da *Symfony* che offre numerose funzionalità legate all'invio delle *e-mail*.

Il *bundle* utilizza alcune variabili d'ambiente definite nel file `.env.local`: la variabile `MAIL_URL`, per esempio, definisce il *server* da utilizzare per l'invio delle *e-mail*. Il contenuto dell'*e-mail*, inoltre, viene salvato in un *template* (*Twig*) e renderizzato prima dell'invio. Per ottenere, invece, l'immagine contenente il codice *QR*, si è creato un oggetto di tipo `QrCode` (classe fornita dal *bundle* `endroid/qr-code`) fornendo in *input* il contenuto del codice (ricavato dal *DTO* salvato in *cache*). L'immagine rappresentata dal precedente oggetto, infine, è stata trasformata in una stringa codificata in *Base64* affinché possa essere inviata all'interno dell'*e-mail*.

```
public function postEnable2faAction(): UserTwoFaConfiguration
{
    /** @var User $currentUser */
    $currentUser = $this->getUser();

    $userTwoFaConfiguration = $this->getTwoFaManager()->initializeTwoFaForUser($currentUser);

    $userEmailAddress = $currentUser->getEmail();

    $email = new Swift_Message();

    $qrCode = new QrCode($userTwoFaConfiguration->getQrCodeContent());
    $qrCodeImage = $qrCode->writeString();

    // Embedding Dynamic Content
    $attachment = new Swift_Image($qrCodeImage, 'qrCode.png', 'image/png');
    $cid = $email->embed($attachment);

    $email
        ->setFrom($this->getMailerSender(), $this->getMailerSenderName())
        ->setTo($userEmailAddress)
        ->setSubject($this->getTranslator()->trans('Two-Factor Authentication activation'))
        ->setBody(
            $this->renderView(
                'CRUD/Emails/two_fa_activation.html.twig',
                [
                    'user' => $currentUser,
                    'backUpCodes' => $userTwoFaConfiguration->getBackupCodeCollection(),
                    'qrCode' => new QrCode($userTwoFaConfiguration->getQrCodeContent()),
                    'cid' => $cid,
                ]
            ),
            'text/html'
        );

    $this->getMailer()->send($email);

    return $userTwoFaConfiguration;
}
```

Listing 6: Metodo `postEnable2faAction`

L'endpoint `codeValidation`, d'altro canto, viene gestito dal metodo `postCodeValidationAction` (*Listing 7*). Quest'ultimo ottiene il codice temporaneo e il nome del documento in *cache* contenente il *DTO* (entrambi contenuti all'interno del *form* compilato e inviato dall'utente). Dopo aver fatto ciò, il metodo esegue la seconda fase dell'attivazione del *2FA* richiamando `validateAndCompleteTwoFaForUser` del componente `TwoFaManager` (utilizzando l'utente in questione come parametro).

```
public function postCodeValidationAction(Request $request): Response
{
    /** @var User $currentUser */
    $currentUser = $this->getUser();

    $hash = $request->request->get('hash');
    $code = $request->request->get('twoFaCode');

    try {
        $this->getTwoFaManager()->validateAndCompleteTwoFaForUser($hash, $code, $currentUser);
    } catch (UserTwoFaConfigurationNotFound $exception) {
        throw new BadRequestHttpException("Invalid hash provided");
    } catch (InvalidTwoFactorCodeException $exception){
        throw new BadRequestHttpException("Invalid code provided");
    }

    return $this->handleView($this->view(null, Response::HTTP_NO_CONTENT));
}
```

Listing 7: Metodo `postCodeValidationAction`

Il metodo `postDisable2faAction` (*Listing 8*), infine, gestisce l'endpoint `disable2fa`: ricava l'utente autenticato che ha richiesto la disattivazione del *2FA* e richiama il metodo `disableFull2FaForUser` del componente `TwoFaManager` utilizzando come parametro l'oggetto utente precedentemente determinato.

```
public function postDisable2faAction(): Response
{
    /** @var User $currentUser */
    $currentUser = $this->getUser();

    $this->getTwoFaManager()->disableFull2FaForUser($currentUser);

    return $this->handleView($this->view(null, Response::HTTP_NO_CONTENT));
}
```

Listing 8: Metodo `postDisable2faAction`

5.4 Creazione del bundle

Nelle precedenti sezioni si è implementato con successo un servizio di autenticazione a due fattori le cui funzionalità possono essere usufruite sia da applicazioni web (tramite *web browser*) sia da applicazioni *mobile* (tramite *REST API*). È necessario, pertanto, progettare un *bundle* che racchiuda il servizio per consentirne l'utilizzo su qualunque applicazione.

In *Symfony*, è consigliato seguire alcune convenzioni e regole durante la creazione di un *bundle* (soprattutto se l'obiettivo è la progettazione di un *bundle* configurabile ed estendibile). Un *bundle*, essendo anche un *namespace*, deve seguire le regole definite dallo standard *PSR-4* per i *namespace* e i nomi delle classi *PHP*. Il *namespace*, perciò, dovrà contenere:

- una parte iniziale che identifichi il produttore (*vendor*)
- zero o più parti che descrivono la categoria a cui appartiene il *bundle*
- una parte finale che identifica il *namespace* e che deve terminare in **Bundle**

Un *namespace* diventa un *bundle* quando a quest'ultimo viene associata una *bundle class*. Quest'ultima deve seguire le seguenti regole:

- utilizzare solo caratteri alfanumerici e *underscore*
- il nome segue il formato *camelCase* con la prima lettera in maiuscolo
- il nome deve essere appropriato e breve (non più di due parole)
- il nome deve essere prefissato dalla parte identificante il produttore (*vendor*) e dalle parti associate alla categorie
- il nome deve terminare con la parola **Bundle**

Nel caso del servizio di autenticazione a due fattori, il *namespace* associato sarà `Cogitoweb\TwoFaBundle` mentre il nome della *bundle class* sarà `CogitowebTwoFaBundle` (*Listing 9*).

```
<?php
namespace Cogitoweb\TwoFaBundle;

use Symfony\Component\HttpKernel\Bundle\Bundle;

class CogitowebTwoFaBundle extends Bundle
{
}
```

Listing 9: *File CogitowebTwoFaBundle.php*

Per quanto riguarda la struttura, il *bundle* dovrà essere progettato seguendo la seguente suddivisione dei *file* e delle cartelle:

```
| - src/
    | - Controller/
    | - DependencyInjection/
    | - Dto/
    | - Entity/
    | - Exception/
    | - Form/
```

```

|- Manager/
|- Message/
|- Model/
|- Resources/
    |- config/
        |- doctrine-mapping/
        |- routing/
        |- serializer/
        |- validator/
        |- services.yaml
    |- translations/
    |- views/
|- CogitowebTwoFaBundle.php
|- composer.json

```

Il *file* `composer.json`, in particolare, dovrà essere progettato come illustrato nel *Listing* 10. I campi di maggior rilievo che compaiono nel *file* sono:

- **name**, composto dal nome del fornitore e dal nome del *bundle* utilizzando i trattini (*hyphen*) come separatori
- **type**, identifica il tipo del progetto (permettendo a *Symfony Flex* di eseguire azioni specifiche)
- **autoload**, utilizzato da *Symfony* per caricare le classi del *bundle*. È consigliato adottare lo standard *PSR-4*, ovvero utilizzare il *namespace* come chiave (*key*) e il percorso in cui si trova la *bundle class* come valore
- **require**, contiene una lista di tutte le dipendenze del servizio (per brevità sono state riportate solo alcune librerie associate al servizio)

Le cartelle all'interno di `src/`, invece, dovranno contenere tutte le classi associate al servizio divise per tipologia (*Controller*, *Dto*, *Entity*, *Exception*, ecc.). La cartella **Resources**, nello specifico, dovrà contenere:

- una cartella **translations** (contenete i *file* che consentono di tradurre le stringhe utilizzate dal servizio in base all'utente collegato)
- una cartella **views** (contenete i *template* utilizzati dal *bundle*)
- una cartella **config** che conterrà:
 - una cartella **doctrine-mapping** (contenente le mappature *Doctrine* delle entità del *bundle*)
 - una cartella **routing** (contenente le rotte definite dal *bundle*)
 - il *file* **services.yaml** (contenente i servizi e i parametri offerti dal *bundle*)
 - una cartella **serializer** (contenente i *file* che indicano come serializzare le entità del *bundle*)
 - una cartella **validator** (contenente i *file* che specificano i vincoli che le entità del *bundle* devono rispettare)


```

{
    "name": "cogitoweb/two-fa-bundle",
    "description":
        "adds logic to implement two-factor authentication in your application",
    "type": "symfony-bundle",
    "license": "proprietary",
    "authors": [
        {
            "name": "Lorenzo D'Antoni",
            "email": "lorenz.dantoni@gmail.com"
        }
    ],
    "require": {
        "php": ">=7.4",
        "doctrine/doctrine-bundle": "^1.12.13",
        "google/cloud-firestore": "^1.19",
        "scheb/2fa-backup-code": "^5.12",
        "scheb/2fa-bundle": "^5.12.0",
        "scheb/2fa-google-authenticator": "^5.12.0",
        "sroze/messenger-enqueue-transport": "^0.5.0",
        "symfony/form": "^4.3",
        "symfony/messenger": "4.4.*"
    },
    "autoload": {
        "psr-4": {
            "Cogitoweb\\TwoFaBundle\\" : "src/"
        }
    }
}

```

Listing 10: *File composer.json*

Dovendo creare un *bundle* (e non una libreria), non è sufficiente fornire le classi legate al servizio *2FA*. È previsto, infatti, che un *bundle*, durante la fase di installazione, si occupi anche di registrare i servizi (associati alle classi del progetto) all'interno del *container* di *Symfony*.

Per prima cosa, è necessario creare il *file* `DependencyInjection/Configuration.php` (*Listing 11*), responsabile dell'analisi e della convalida dei *file* di configurazione del *bundle*.

```

class Configuration implements ConfigurationInterface
{
    public function getConfigTreeBuilder(): TreeBuilder
    {
        $treeBuilder = new TreeBuilder();
        $treeBuilder->root('cogitoweb_two_fa');
        return $treeBuilder;
    }
}

```

Listing 11: *File Configuration*

È richiesta, inoltre, la creazione del *file* `DependencyInjection/CogitowebTwoFaExtension.php` (*Listing 12*) per effettuare il caricamento dei *file* di configurazione e la registrazione dei servizi del *bundle*.

```

class CogitowebTwoFaExtension extends Extension
{
    public function load(array $configs, ContainerBuilder $container)
    {
        $configuration = new Configuration();
        $this->processConfiguration($configuration, $configs);

        $loader = new YamlFileLoader(
            $container,
            new FileLocator(__DIR__ . '/../Resources/config')
        );
        $loader->load('services.yaml');
    }
}

```

Listing 12: *File CogitowebTwoFaExtension.php*

Symfony, durante la costruzione del *container*, per ogni *bundle* installato cerca (all'interno della cartella `DependencyInjection`) una classe con lo stesso nome del *bundle* ma che termina in *Extension*. Se la classe esiste, si procederà all'istanziatura di quest'ultima e verrà richiamato il metodo *load*, responsabile di caricare il file `services.yaml` (contenente tutti i servizi del *bundle*).

Per importare le rotte definite dal *bundle*, invece, è sufficiente aggiungere le seguenti righe al file `routes.yaml` dell'applicazione web:

```

2fa_cache_cleanupper:
    resource: '@CogitowebTwoFaBundle/Resources/config/routing/two_fa.yaml'

```

I *template* e le traduzioni, infine, sono importati automaticamente. L'unica accortezza da seguire è la struttura del *bundle* definita precedentemente.

Fatto ciò, il *bundle* potrà essere utilizzato da qualsiasi piattaforma web. In questo senso, è necessario modificare il file `composer.json` dell'applicazione web in questione: il campo `repositories` dovrà contenere un nuovo oggetto che segnali a *Symfony* dove trovare il codice sorgente del *bundle*. In questo caso si è deciso di caricare il progetto del *bundle* su *GitLab*, pertanto dovrà essere specificato l'indirizzo *URL* del *repository* associato.

Per installare e testare il *bundle* in locale, infine, è sufficiente utilizzare il seguente comando:

```
> composer require "cogitoweb/two-fa-bundle @dev"
```

Il *flag* `@dev` è uno *stability flag*. Questa particolare tipologia di *flag* permette di restringere o espandere la stabilità di un *bundle*. I *flag*, inoltre, consentono di non dover specificare una versione del *bundle*: in questo frangente, infatti, si è precisato che il *bundle* è ancora in fase di sviluppo.

6

Conclusioni

L'obiettivo della tesi e dell'attività di tirocinio riguardava la progettazione di un servizio di autenticazione a due fattori (*2FA*) da poter utilizzare all'interno di una piattaforma web. Nei precedenti capitoli sono state delineate tutte le fasi necessarie alla creazione di un servizio *2FA* che sia conforme ai requisiti e alle specifiche imposti dal progetto in questione.

Il servizio implementato consente agli utenti di un'applicazione web di attivare un secondo fattore di autenticazione da utilizzare nel corso della procedura di *login*. Le funzionalità del sistema, inoltre, sono state rese utilizzabili anche dalle applicazioni *mobile* tramite la progettazione di una *REST API*.

In merito alla tipologia di *2FA*, si è scelto di adottare la modalità basata su *token software* (utilizzando *TOTP* come algoritmo di generazione dei codici temporanei) in quanto risulta un buon compromesso tra sicurezza e usabilità. Durante l'attivazione del *2FA*, inoltre, vengono forniti all'utente 10 codici monouso da utilizzare in caso di imprevisti riguardanti il secondo fattore (procedura di *backup*). Il processo di attivazione, peraltro, è stato suddiviso in due fasi separate per permettere all'utente di confermare esplicitamente la propria scelta.

Si è cercato, per quanto possibile, di disaccoppiare i componenti del servizio dai moduli del sistema in cui verrà aggiunto il *2FA*. La progettazione di un servizio modulare con un basso livello di accoppiamento con il sistema esterno ha permesso la realizzazione di un *bundle* che racchiuda le funzionalità offerte dal servizio, consentendone l'utilizzo su qualsiasi applicazione web che utilizzi *Symfony* come *framework*. Il servizio, oltretutto, è stato reso sufficientemente robusto da poter gestire in modo adeguato e *ragionevole* alcune situazioni impreviste o anomale.

Per quanto riguarda l'interfaccia utente, ogni applicazione web può definire i propri *template*. Il *bundle* fornisce, tuttavia, dei *template* di *default* la cui progettazione è stata guidata da due parole chiave: **semplicità** ed **efficacia**. La maggior parte delle persone, come spiegato nei capitoli precedenti, ritiene che il tempo e la complessità aggiunti dalle procedure di *2FA* non valgano veramente la sicurezza aggiuntiva. Si è scelto di adottare, dunque, procedure rapide ed intuitive.

Il servizio sviluppato, tuttavia, è sicuramente migliorabile: l'obiettivo principale era quello sviluppare le *fondamenta* di un servizio di autenticazione a due fattori. Prima di rilasciare il servizio su vasta scala, infatti, è opportuno sviluppare una soluzione contro gli attacchi di tipo *brute force*. È possibile limitare il numero di codici temporanei inseribili dall'utente in diversi modi:

- tra un tentativo e l'altro introdurre degli intervalli temporali in cui non è possibile effettuare l'autenticazione
- bloccare temporaneamente l'*account* dell'utente dopo un certo numero di tentativi di autenticazione falliti
- aumentare la complessità della procedura di autenticazione inserendo, per esempio, un *CAPTCHA* (*Completely Automated Public Turing test to tell Computers and Humans Apart*)

Tuttavia, essendo il codice temporaneo di 6 cifre, un attaccante dovrebbe cercare il codice corretto tra 1 milione di possibili codici in una finestra temporale di 30 secondi (questo perché un nuovo codice viene generato ogni 30 secondi). In media, quindi, si dovrebbero tentare circa 500000 codici in un intervallo di 30 secondi: 0.06 millisecondi a codice. Dato che il server non consente di validare un codice in un intervallo di tempo così ristretto, si è deciso, almeno inizialmente, di non adottare una soluzione contro gli attacchi *brute force*.

Il database *NoSQL* utilizzato (*Firestore*), inoltre, non è principalmente rivolto al *caching*. Se il *bundle 2FA* dovesse essere utilizzato in applicazioni web mediamente grandi, è opportuno adottare dei database progettati esclusivamente per il *caching* (per esempio *Memorystore*) vista l'elevata scalabilità e la bassa latenza offerti da quest'ultimi.

L'aggiunta, infine, di più modalità di autenticazione a due fattori (*token software*, *token hardware*, *push-based*, ecc.) è certamente un miglioramento che bisognerà pianificare. Molte aziende, infatti, già lo consentono in quanto, la possibilità di scegliere la modalità con cui effettuare l'autenticazione, facilita l'adozione del *2FA* da parte degli utenti.

6.1 Uno sguardo al futuro

Nonostante l'autenticazione multifattoriale standard sia una soluzione sicuramente valida nel presente, nei prossimi anni si affermerà sempre di più una particolare modalità di autenticazione che, al posto di definire a priori le informazioni che un utente deve inserire, consente di richiedere diversi tipi di credenziali in base alla specifica situazione: si tratta dell'**autenticazione adattiva**.

È possibile creare un profilo per ogni utente contenente tutte le informazioni che possono risultare utili durante l'autenticazione (la posizione geografica dell'utente, i dispositivi associati a quell'utente, il suo ruolo aziendale, ecc.). Quando viene effettuato un tentativo di *login*, la richiesta viene analizzata e le viene associato un livello di rischio. In base al punteggio ottenuto, all'utente verrà richiesto di fornire credenziali aggiuntive o, al contrario, gli sarà consentito di effettuare il *login* inserendo meno informazioni.

La maggior parte delle implementazioni di questa modalità di autenticazione utilizzano algoritmi di *Machine Learning*. Quest'ultimi, con l'avanzare del tempo, controllano ed imparano il comportamento dell'utente per creare un profilo accurato dei suoi *pattern* di *login*. È possibile, quindi, memorizzare i dispositivi utilizzati dagli utenti, gli orari in cui viene solitamente effettuato il *login*, i luoghi di lavoro più frequenti, indirizzi *IP* e così via. In questo modo si possono identificare delle anomalie nei *pattern* di *login* e adeguare, di conseguenza, le credenziali da richiedere agli utenti o addirittura negare l'accesso.

Oltre a fornire maggiore sicurezza, l'autenticazione adattiva agevola il processo di *login*: l'autenticazione multifattoriale richiede ad un utente di inserire, in qualsiasi situazione, tutte le credenziali previste mentre l'autenticazione adattiva può richiedere meno informazioni ad un utente il cui comportamento viene riconosciuto. Ciò comporta una maggior sicurezza e, allo stesso tempo, uno snellimento delle procedure di autenticazione.

Bibliografia

- [1] Direttiva (UE) 2015/2366 del Parlamento Europeo e del Consiglio del 25 novembre 2015 relativa ai servizi di pagamento nel mercato interno, che modifica le direttive 2002/65/CE, 2009/110/CE e 2013/36/UE e il regolamento (UE) n. 1093/2010, e abroga la direttiva 2007/64/CE, [2015].
- [2] Fido U2F Complete v1.2 PS 20170411. Relazione tecnica, FIDO Alliance, 2017.
- [3] Businesses at Work. Relazione tecnica, Okta, 2021. <https://www.okta.com/sites/default/files/2021-03/Businesses-at-Work-2021.pdf>.
- [4] Google Cloud. What is Pub/Sub? <https://cloud.google.com/pubsub/docs/overview>. [Ultimo accesso: 7-12-2021].
- [5] Roy T. Fielding e Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Authentication. RFC 7235, giugno 2014.
- [6] Paul A. Grassi, James L. Fenton, Elaine M. Newton, Ray A. Perlner, Andrew R. Regenscheid, William E. Burr, e Justin P. Richer. Digital Identity Guidelines - Authentication and Lifecycle Management. Relazione Tecnica NIST Special Publication 800-63B, Includes updates as of March 02, 2020, National Institute of Standards and Technology, 2017.
- [7] Paul A. Grassi, Michael E. Garcia, e James L. Fenton. Digital Identity Guidelines. Relazione Tecnica NIST Special Publication 800-63-3, Includes updates as of March 02, 2020, National Institute of Standards and Technology, 2017.
- [8] Emin Huseynov e Jean-Marc Seigneur. Capitolo 50 - Context-Aware Multifactor Authentication Survey In *Computer and Information Security Handbook (Third Edition)*. A cura di John R. Vacca, pp. 715–726. Morgan Kaufmann, Boston, terza edizione, 2017.
- [9] Assaf Klinger. Technical report on SS7 vulnerabilities and mitigation measures for digital financial services transactions. Relazione tecnica, FIGI (Financial Inclusion Global Initiative), 2020.
- [10] National Institute of Standards e Technology. Security requirements for cryptographic modules. Relazione Tecnica Federal Information Processing Standards Publications (FIPS PUBS) 140-2, Change Notice 2 December 03, 2002, U.S. Department of Commerce, 2001.
- [11] Ken Reese, Trevor Smith, Jonathan Dutson, Jonathan Armknecht, Jacob Cameron, e Kent E. Seamons. A Usability Study of Five Two-Factor Authentication Methods In *Fifteenth Symposium on Usable Privacy and Security, SOUPS 2019, Santa Clara, CA, USA, August 12-13, 2019*. A cura di Heather Richter Lipford. USENIX Association, 2019.

- [12] Kelley Robinson. What is a Time-based One-time Password (TOTP)? <https://www.twilio.com/docs/glossary/totp>. [Ultimo accesso: 03-11-2021].
- [13] Christian Schieb. SchiebTwoFactorBundle: The Authentication Process with Two-Factor Authentication. <https://symfony.com/bundles/SchiebTwoFactorBundle/6.x/index.html>. [Ultimo accesso: 11-01-2022].
- [14] Kurt Thomas e Angelika Moscicki. How effective is basic account hygiene at preventing hijacking. Relazione tecnica, Google, New York University, University of California, 2019.
- [15] Wikipedia. Oggetto di trasferimento dati. https://it.wikipedia.org/wiki/Oggetto_di_Trasferimento_Dati. [Ultimo accesso: 31-12-2021].