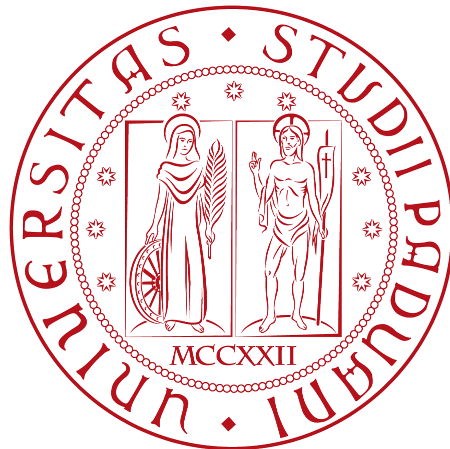
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Implementazione JWT per messa in sicurezza di un API tramite Spring Security

Tesi di laurea

Relatore

Prof. Francesco Ranzato

Laureando

Mattia Casarotto

Anno Accademico 2022-2023

Mattia Casarotto: *Implementazione JWT per messa in sicurezza di un API tramite Spring Security,* Tesi di laurea, © Settembre 2023.

Sommario

Il seguente documento descrive la mia esperienza di stage svolta tra il 10/07/2023 e il 01/09/2023 presso l'azienda Sync Lab s.r.l. della durata approssimativa di 320 ore. L'esperienza di stage si è divisa in parti prettamente di studio individuale e una fase di implementazione effettiva. L'obiettivo dell'esperienza è stato quello di mettere in sicurezza una Web Application utilizzando Spring Security e, in particolare, sfruttando la tecnologia divenuta ormai standard JSON Web Token (JWT).

Ringraziamenti

Innanzitutto, vorrei ringraziare il Prof. Francesco Ranzato, relatore della mia tesi, per la disponibilità e l'aiuto fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori per essermi stati vicini ed aver creduto in me nel corso di tutti questi anni.

Ringrazio infine i miei amici per il sostegno dimostratomi e per le esperienze che mi hanno regalato durante il mio percorso.

Padova, Settembre 2023 Mattia Casarotto

Indice

1 Introduzione **1**
1.1 Azienda 1
1.2 Organizzazione del lavoro 1
1.3 Tecnologie affrontate 2
1.4 Strumenti di comunicazione 2
1.5 Obiettivi prefissati 3
1.6 Distribuzione temporale 4
1.6.1 Diagramma di Gantt 4

2 Il progetto **7**
2.1 Introduzione 7
2.2 L'idea 7
2.3 La base di dati 7
2.4 Servizi 8

3 Protocolli di sicurezza **9**
3.1 Introduzione 9
3.2 Protocollo HTTPS 9
3.3 Protocollo OAuth 1.0 11
3.4 Protocollo OAuth 2.0 11
3.4.1 Ruoli 12
3.4.2 Bearer Token 13
3.4.3 Authorization Grant 13
3.5 OIDC 15

4 Sicurezza nel progetto **17**
4.1 Da API individuali a gateway 17
4.2 Spring Cloud 17
4.3 JWT 18
4.3.1 Claims 19
4.3.2 Refresh Token 19
4.4 Authentication Flow 20
4.5 Blacklist 21
4.6 Intercettare e modificare il response body 21
4.7 CORS 21
4.8 Spring Security 22
4.8.1 Security Context Holder 22
4.8.2 CSRF 23

5 Implementazione OAUTH2 tramite Keycloak **25**
5.1 Introduzione 25
5.2 Scelta dell'Authorization Server 25
5.3 Struttura Keycloak 26
5.4 Integrazione all'interno del progetto 26
5.5 Considerazioni finali 27

6 Implementazioni sul front-end **29**
6.1 Introduzione 29
6.2 HttpInterceptor 29
6.3 Guard 30

7 Verifica e validazione **31**
7.1 Introduzione 31
7.2 Strumenti utilizzati 31
7.3 Code coverage conseguita 31
7.3.1 Gateway 31
7.3.2 Security servizi 32
7.4 Collaudi e validazione 32

8 Conclusioni **33**
8.1 Obiettivi Raggiunti 33
8.2 Conoscenze acquisite 33
8.2.1 Protocolli di sicurezza 33
8.2.2 Progettazione 33
8.2.3 Way of Working 34
8.3 Evoluzione rispetto a consuntivo 34
8.4 Valutazione personale 34

Acronimi e abbreviazioni **35**

Glossario **37**

Bibliografia **43**

Elenco delle figure

1.1 Integrazione spring-angular 3
1.2 Estratto del diagramma di Gantt 5

2.1 Diagramma di TripHippie 8

3.1 Una semplice comunicazione HTTP 10
3.2 La differenza tra una richiesta tradizionale di tipo HTTP e una di tipo HTTPS 10
3.3 Differenze tra gli authentication flow di OAUTH 1 e OAUTH 2 12
3.4 Ruoli in OAUTH 2 13
3.5 Authorization Code flow 14
3.6 Device Flow 15
3.7 OpenID viene utilizzato in modo complementare ad OAUTH 2 16

4.1 Un estratto delle routes del Gateway, configurate tramite Bean 18
4.2 Un esempio del contenuto di un JWT 19
4.3 Richiesta di refresh 20
4.4 Esempio di Authentication Flow all'interno del progetto TripHippie . . 20
4.5 Configurazione della filter chain all'interno di UserService 22
4.6 E' sufficiente inserire l'authentication negli input della funzione per costruirla a partire dal contesto 23
4.7 Funzionamento di un attacco CSRF 23

5.1 Logo Keycloak 26
5.2 SSO tramite Keycloak 27

6.1 Il flow di una http request in Angular con l'utilizzo di un HttpInterceptor 29
6.2 Il funzionamento di un Guard in breve 30

7.1 Code Coverage package security 31
7.2 Code coverage package security 32
7.3 Code coverage package filter 32

Elenco delle tabelle

1.1 Distribuzione ore settimanale 4

Capitolo 1

Introduzione

1.1 Azienda

Sync Lab nasce nel lontano 2002 a Napoli come Software House e si è rapidamente affermata nel panorama IT italiano, vantando a oggi 6 sedi e oltre 150 clienti, tra cui alcune tra le più grandi compagnie italiane quali **Vodafone**, **Tim**, **Sky** e **Unicredit**. L'azienda propone soluzioni riguardanti svariati mercati quali sanità, industria, finanza, trasporti e si mantiene all'avanguardia seguendo nuove tendenze e tecnologie tra cui **Big Data**, **Cloud Computing**, **IOT** e **Software House**.
Ho conosciuto l'azienda tramite un incontro telematico conoscitivo svoltosi poco dopo l'evento StageIT 2023 con l'ingegnere che sarebbe poi stato il mio tutor durante l'esperienza, ossia Fabio Pallaro. La mia scelta si è basata principalmente su motivazioni riguardanti le tecnologie scelte, molto utilizzate e quindi utili successivamente in contesti lavorativi, e su motivi logistici, in quanto l'azienda mi ha offerto ampio utilizzo di Smart Working e l'accesso a una sede raggiungibile senza eccessiva fatica.

1.2 Organizzazione del lavoro

Il lavoro è stato svolto utilizzando la metodologia **Agile**, cioè ponendo massima enfasi sul software piuttosto che sulla documentazione; per raggiungere questo obiettivo il lavoro è stato suddiviso in brevi cicli iterativi e incrementali - tendenzialmente della durata di una settimana - al termine delle quali veniva svolto uno Stato Avanzamento Lavori, ovvero un meeting (svolto tramite Google Meet per via della differenza di sedi dei diversi membri del progetto) dove raccogliere feedback e decidere come procedere per il prossimo ciclo.

1.3 Tecnologie affrontate

Nel corso dello stage, sono state utilizzate le seguenti tecnologie per raggiungere il completamento del progetto del tirocinio:

- **Javascript**: linguaggio di programmazione orientato agli eventi utilizzato sia lato Client web sia Server, nel caso di questo stage lato client (AngularJS);

- **HTML**: Linguaggio di markup utilizzato per l'impaginazione della pagina web Angular;

- **CSS**: Linguaggio utilizzato per formattare documenti come pagine HTML o XML descrivendone la resa grafica;

- **Java**: Linguaggio ad alto livello orientato agli oggetti, fondamentale per la realizzazione del Back-End;

Sono stati inoltre utilizzati due framework fondamentali:

- **Spring**: Si tratta di un framework open source per sviluppo di applicazioni Java. Tra le sue caratteristiche fondamentali troviamo l'utilizzo di **Dependency Injection** e l'offerta, tra le varie parti che lo formano, di:
  - **Spring Boot**: Strumento che semplifica lo sviluppo di applicazioni web permettendo di minimizzare le attività di configurazione;
  - **Spring Data**: Permette di implementare repository basate su JPA, velocizzando e semplificando le operazioni CRUD su database;
  - **Spring Security**: Framework che garantisce autenticazione e autorizzazione ai nostri applicativi Java.

- **Angular**: Framework per costruire web applications per piattaforme sia mobile sia desktop.

1.4 Strumenti di comunicazione

Per coordinare il gruppo di lavoro e comunicare tra di noi sono stati utilizzati principalmente 3 strumenti, ossia:

- **Google Meet**: Applicazione di teleconferenze, utilizzata abbastanza frequentemente per colloqui individuali o per incontri di gruppo con membri del personale che non riescono ad essere presenti in giorno, situazione abbastanza comune in quanto l'azienda fa ampio uso di smartworking;

- **Discord**: Piattaforma di instant messaging, utilizzata molto frequentemente per chiedere e/o ottenere informazioni veloci e coordinarsi per le giornate in presenza;

- **Trello**: Strumento di gestione di progetti, permette la creazione e gestione semplificata di bacheche per mantenere le task organizzate.

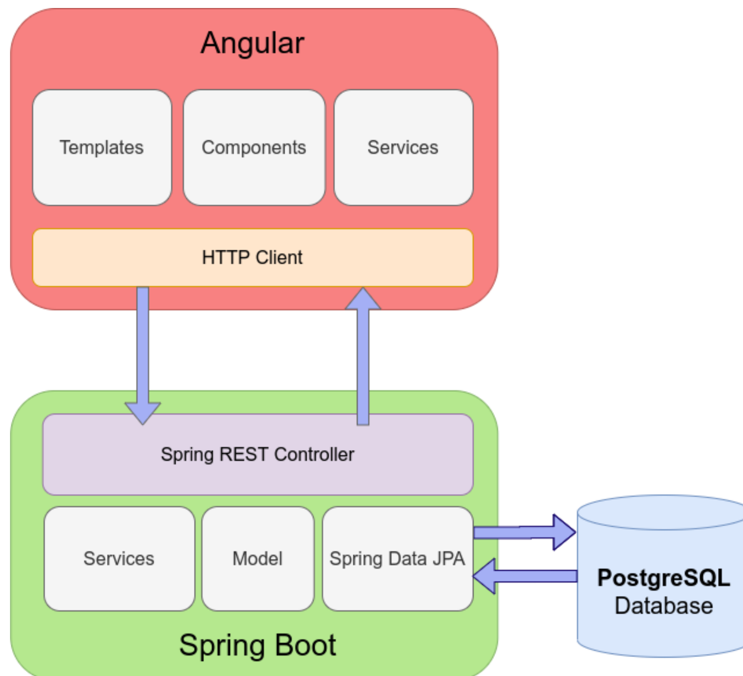


Figura 1.1: Integrazione spring-angular
Source: FrontBackend.com

1.5 Obiettivi prefissati

L'attività dello stage si è fondalmentalmente divisa in due parti, una prettamente di studio individuale e familiarizzazione con le tecnologie da utilizzare successivamente e l'altra di implementazione vera e propria. Nella fase di implementazione, in particolare, mi è stato richiesto di implementare la security all'interno del nostro progetto, utilizzando JWT in combinazione con protocolli a mia scelta. L'azienda ha fissato i seguenti obiettivi per il successo di questa attività di stage didattico:

- Obbligatori (**Percentuale raggiunta: 100%**)
  - **O01**: Acquisizione competenze sulle tematiche sopra descritte;
  - **O02**: Capacità di raggiungere gli obiettivi richiesti in autonomia seguendo il cronoprogramma;
  - **O03**: Portare a termine le implementazioni previste con la messa in sicurezza delle funzionalità. di login/gestione viaggi per la parte di Back-End;
- Desiderabili (**Percentuale raggiunta: 100%**)
  - **D01**: Portare a termine le implementazioni previste con la messa in sicurezza di tutte le funzionalità anche per la parte di Front-End;
- Facoltativi (**Percentuale raggiunta: 100%**)
  - **F01**: Dare un contributo proattivo sui dettagli implementativi in merito al meccanismo di sicurezza da implementare (Refresh Token o altre soluzioni architetturali).

1.6 Distribuzione temporale

Il progetto si è svolto lungo un lasso temporale di 8 settimane (320 ore), con la seguente ripartizione di attività per iterazione:

Attività per iterazione		
Settimana	**Attività svolte**	**Ore**
1	Formazione base	40
	Incontro con persone coinvolte nel progetto per discutere i requisiti e le richieste relativamente al sistema da sviluppare Verifica credenziali e strumenti di lavoro assegnati Ripasso Java Standard Edition e tool di sviluppo (IDE, StopLight) Studio teorico dell'architettura a Microservizi: passaggio da Monolite ad architetture a Microservizi con pro e contro Ripasso principi della buona programmazione (SOLID, CleanCode)	
2	Formazione Microservizi, Spring Core/Spring Boot e ORM	40
	Studio teorico dell'architettura a microservizi: API Gateway, Service Discovery e Service Registry, Circuit Breaker e Saga Pattern Studio Spring Core/Spring Boot Studio ORM, in particolare il framework Spring Data JPA	
3	Formazione REST e security	40
	Studio servizi REST e framework Spring Data REST Studio meccanismi di securizzazione web (lato applicativo), framework Spring Security e realizzazione mini prototipo	
4	Analisi e prima implementazione su back end	40
	Analisi dell'idea architetturale attuale di TripHippie Implementazione del meccanismo JWT lato back end con Spring Security sul progetto TripHippie	
5	Front end Angular	40
	Studio linguaggio Typescript e framework Angular Realizzazione mini prototipo front end con angular	
6	Implementazione del meccanismo di sicurezza JWT sulla parte front end attuale del progetto Trip Hippie	40
7	Termine sviluppi ed integrazione	40
8	Collaudi	40

Tabella 1.1: Distribuzione ore settimanale

1.6.1 Diagramma di Gantt

Sulla base di questa distribuzione, inoltre, ho realizzato un diagramma di Gantt, di cui riporto un estratto:

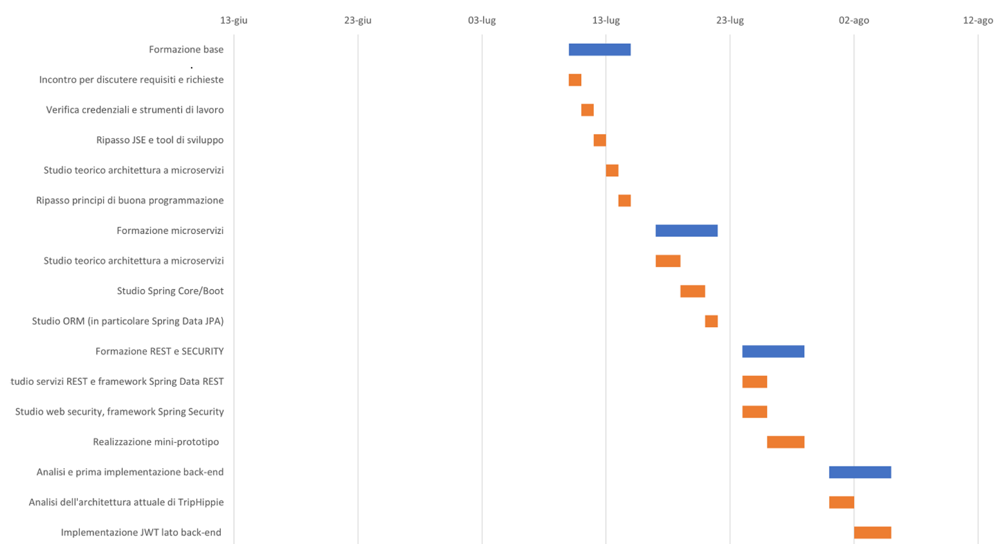


Figura 1.2: Estratto del diagramma di Gantt

Capitolo 2

Il progetto

2.1 Introduzione

In questa sezione descriverò brevemente il processo che ha portato alla creazione del progetto TripHippie e le parti del progetto che non sono state gestite da me in modo individuale. Lo scopo di questa sezione è quello di dare un minimo di visione d'insieme sul progetto nella sua interezza, in modo da rendere più chiare le necessità della Security dal punto di vista implementativo.

2.2 L'idea

Il progetto nasce da un'idea di un ingegnere presso l'azienda, Daniele Zorzi, e consiste in una piattaforma dove appassionati di viaggi possano riunirsi per trovare persone simili a loro e pianificare - o unirsi a - viaggi nel modo più semplice possibile. Il concetto chiave, quindi, è quello dei viaggi: pianificarli, crearli, e modificarne i dettagli tempestivamente e con comodità, con un sistema di inviti semplice e intuitivo che permetta di monitorare i partecipanti molto facilmente. Poichè si tratterebbe di una piattaforma decisamente social, è stata posta molta enfasi sulla creazione di una grafica accattivante e poco formale che invogli l'utente ad utilizzarla.

2.3 La base di dati

Il primo passo è stato quello di - in gruppo - riunirsi per decidere quello che sarebbe stato necessario per la base di dati del nostro progetto secondo l'approccio top down; il progetto è quindi partito dall'ovvia base di utenti e travel espandendosi a partire da essa. Una volta deciso, è stato creato e condiviso un diagramma in modo che ciascun membro del gruppo avesse chiaro su cosa andare a lavorare. In questa fase del progetto, inoltre, l'azienda ha specificato che avremmo usato un singolo Database almeno durante il progetto, raccomandandoci però di far attenzione a mantenere il tutto scalabile in caso di un passaggio a database multipli (evitando ad esempio di fare affidamento sul cascade per le cancellazioni).

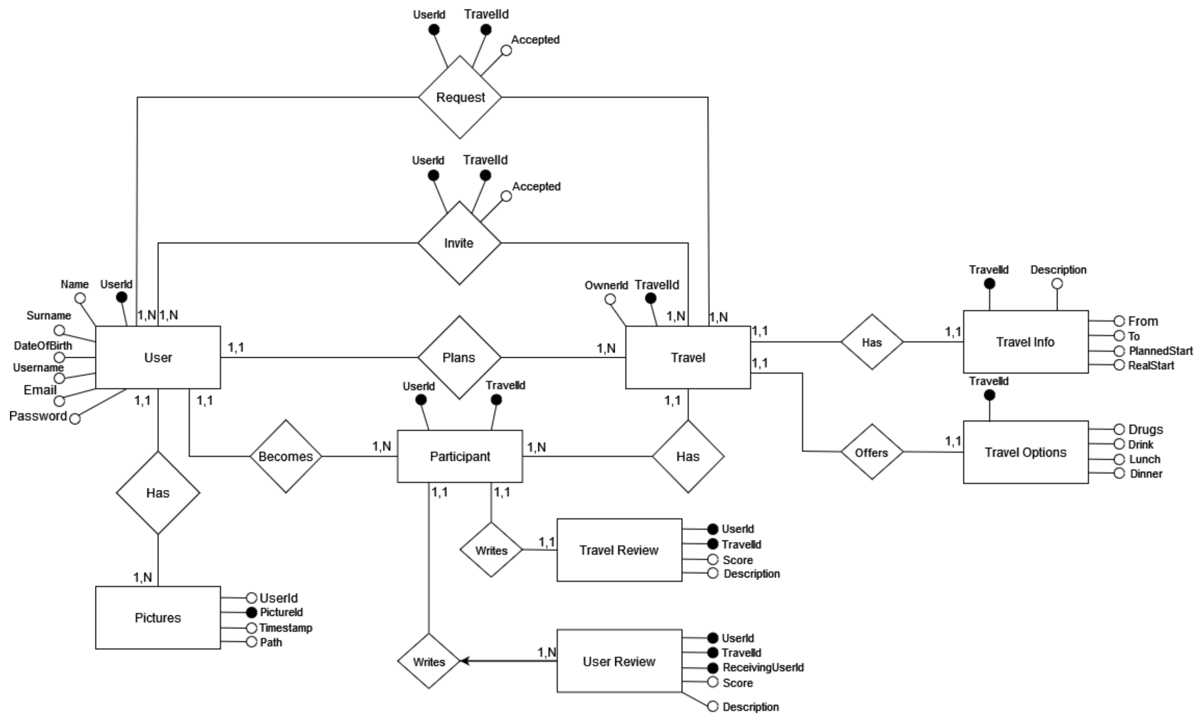


Figura 2.1: Diagramma di TripHippie

2.4 Servizi

Il passaggio successivo è stato quello di decidere i servizi da implementare: per rispondere alle esigenze del progetto la scelta in questa prima fase è ricaduta su 3 API, una per gli utenti e i loro dati personali, una per i viaggi e quello che è a loro collegato e una per le recensioni.

Capitolo 3

Protocolli di sicurezza

3.1 Introduzione

Come già accennato in una sezione precedente, la prima fase dello stage ha consistito fondamentalmente in studio individuale; in particolare, una parte importante di questo studio individuale è stata quella di analizzare i diversi protocolli di sicurezza in modo da avere un quadro completo su quale implementare e cosa lo caratterizza. Di seguito procederò a illustrarne alcuni dei più significativi, alcuni dei quali sono poi stati utilizzati anche per l'implementazione della sicurezza all'interno del progetto.

3.2 Protocollo HTTPS

Il primo protocollo analizzato, banalmente, è stato l'HTTPS. Come si può intuire dal nome, si tratta di un'estensione del protocollo HTTP nato per risolvere alcune delle vulnerabilità del protocollo dal punto di vista della sicurezza (non a caso, infatti, le connessioni con siti web che implementano protocolli di tipo HTTP e non HTTPS vengono automaticamente bollate come "non sicure"). HTTP è un modello di architettura di tipo client-server dove, tendenzialmente, ad assumere il ruolo del client è un browser, mentre quello del server è tendenzialmente assunto dalla macchina dove risiede il sito web. Una volta stabilita la comunicazione tra le due parti viene inviato un messaggio HTTP che consiste, in sostanza, in una serie di righe di testo scritte seguendo il protocollo HTTP, ed è quindi formato da due parti distinte: una richiesta (inviata dal client) e una risposta (ritornata dal server).

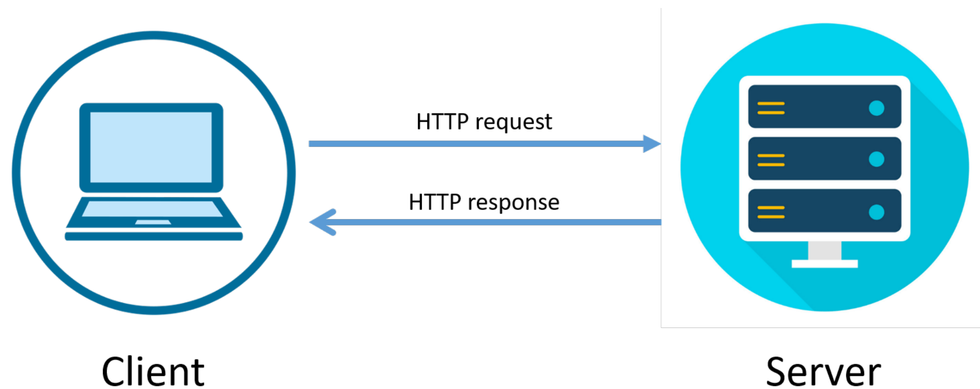


Figura 3.1: Una semplice comunicazione HTTP
Source: BytesOfGigabytes

Il problema è, tuttavia, che queste righe di testo vengono inviate **in chiaro**: questo significa che chiunque stia monitorando la connessione è in grado di leggere i dati inviati tramite il protocollo, e in caso questi dati siano sensibili (ad esempio una password, il numero di una carta di credito o delle informazioni mediche) è facile notare come si tratti di un'enorme falla di sicurezza, rendendo il protocollo estremamente vulnerabile ad attacchi di tipo **Man in the Middle**. In caso il messaggio venga intercettato, infatti, risulterebbe estremamente facile per una figura malintenzionata rubare quei dati all'utente. Per risolvere questo problema HTTPS fa uso di TLS, un protocollo di comunicazione. TLS è il diretto successore di SSL e proprio come il suo predecessore fa uso di certificati digitali che facilitano il processo di handshake ed è in grado di stabilire connessioni crittografate tra un browser e un server web, facendo quindi sì che i messaggi HTTP non siano più in chiaro: in questo modo se anche un attore malintenzionato riuscisse a intercettare la richiesta non sarebbe in grado di leggerne il contenuto, ma otterrebbe semplicemente caratteri incomprensibili.

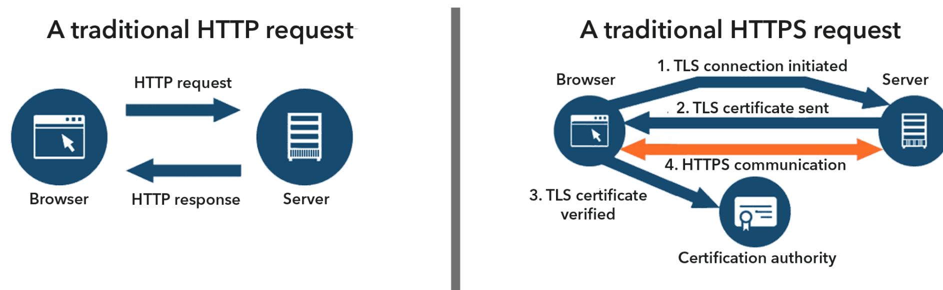


Figura 3.2: La differenza tra una richiesta tradizionale di tipo HTTP e una di tipo HTTPS
Source: fasterize.com

HTTPS, inoltre, è dotato di un'altra funzionalità fondamentale che lo separa da HTTP, cioè la possibilità di autenticare siti web, dando all'utente la certezza che il sito con cui sta provando a stabilire una connessione sia esattamente chi dice di essere, e che

la piattaforma sia quindi degna della propria fiducia. Dove HTTP era fondato su un principio di fiducia, HTTPS implementa certificati digitali che vengono rilasciati e firmati da delle autorità, garantendo l'identità del server. Questo garantisce protezione da alcuni tipi di attacchi, tra cui:

- **On-Path Attacks**: Un attacco informatico dove un soggetto malintenzionato segretamente ritrasmette o altera la comunicazione tra due parti che credono di comunicare direttamente tra di loro;

- **DNS Hijacking**: Un attacco dove gli indirizzi DNS vengono dirottati e il traffico reindirizzato verso falsi server DNS;

- **Domain Spoofing**: Un tipo di attacco informatico cha va a modificare le associazioni tra nomi a dominio e indirizzi IP su un server DNS per reindirizzare i dati verso server controllati dall'attaccante.

3.3 Protocollo OAuth 1.0

OAUTH è un protocollo di sicurezza Open Standard implementabile da qualunque sito. Il concetto fondamentale è quello di utilizzare non le credenziali per autenticare l'utente, ma bensì un access token (tipicamente, un JWT). Questa tecnologia diventa particolarmente utile considerando che, nel panorama attuale, sempre più siti sfruttano funzionalità "esterne" che, per quanto possano risultare utili, pongono un grande problema: condividere la propria identità con questi siti. Il vantaggio principale di questo protocollo di sicurezza è, infatti, quello di consentire l'accesso a diverse risorse senza dover condividere con nessuna di esse le proprie credenziali, ma semplicemente il proprio token, che garantirà un accesso limitato a tale risorsa.

3.4 Protocollo OAuth 2.0

Pur rimanendo un deciso miglioramento sulla semplice autenticazione via username e password, OAUTH 1 presentava diversi problemi, di cui riporto alcuni dei più significativi individuabili nei seguenti punti:

- La necessità di generare credenziali temporanee spesso scartate senza nemmeno venire utilizzate;

- La necessità che tutti gli endpoint abbiano accesso alle credenziali dell'utente in modo da poter autenticare la richiesta;

- La difficoltà per diversi developer abituati alla più semplice autenticazione via username e password di implementare crittografia spesso complessa per ogni richiesta.

Questi problemi hanno eventualmente portato alla nascita del protocollo OAUTH 2 che, seppur basandosi su concetti di base e obiettivi simili a quelli di OAUTH 1, non è una semplice evoluzione di esso ma un vero e proprio nuovo protocollo.

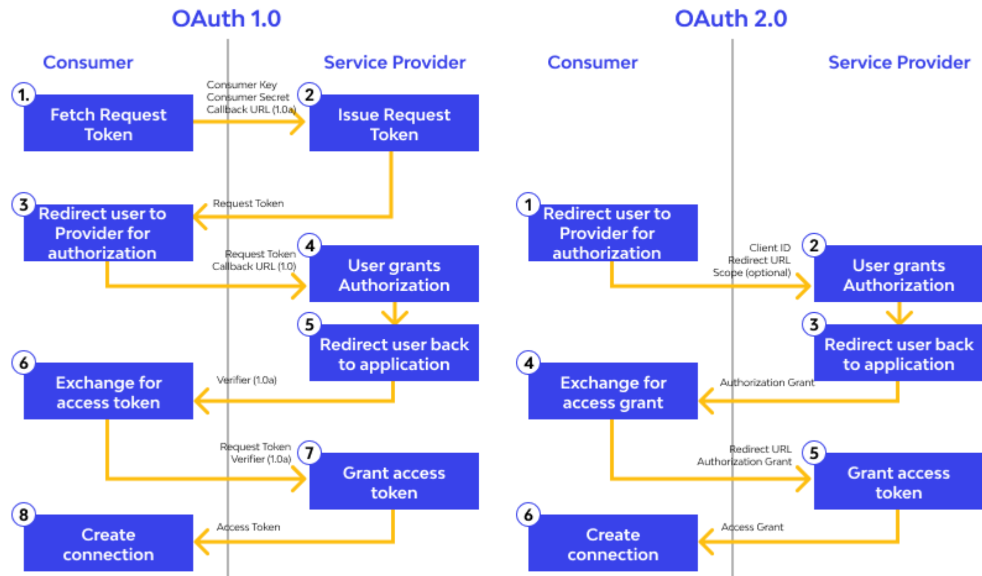


Figura 3.3: Differenze tra gli authentication flow di OAUTH 1 e OAUTH 2
Source: Wallarm.com

3.4.1 Ruoli

Le tre figure di OAUTH 1 vengono abbandonate in favore di 4 figure, ovvero:

- **Authorization Server**: la figura che si occupa di verificare l'identità dell'utente e che effettua il rilascio degli access token allo stesso per poter accedere all'applicazione;

- **Resource Server**: si occupa di gestire richieste autenticate dopo che l'utente ha ottenuto un access token dall'authorization server;

- **Client**: l'applicazione che vuole accedere alle risorse dell'utente;

- **Resource Owner**: l'utente che autorizza l'applicazione ad accedere alle proprie risorse.

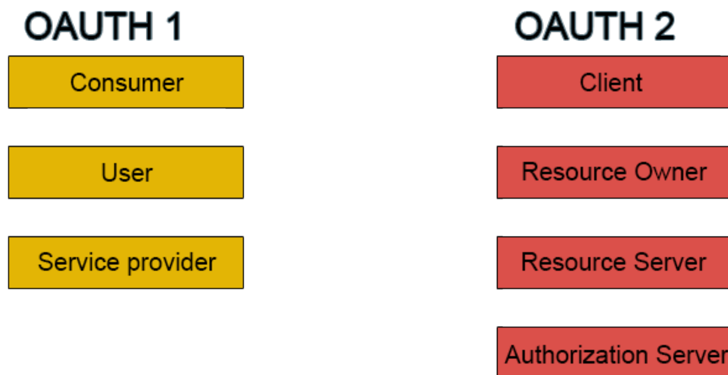


Figura 3.4: Ruoli in OAUTH 2

Al centro del nuovo protocollo OAUTH 2 c'è la volontà di separare interamente la figura dell'API server da quella dell'**Authorization Server**, permettendo che entrambe le parti possano espandersi in modo completamente indipendente. Il **Resource Server** si occuperà, quindi, semplicemente di validare il token, verificando che il client abbia autorizzazioni adeguate per effettuare l'operazione che desidera compiere. Per farlo osserva gli **Scope** associati con il Token, negando la richiesta se gli scope del token non includono quello richiesto per eseguire l'azione richiesta.

3.4.2 Bearer Token

Inoltre, mentre in OAUTH 1 l'access token era formato da una stringa pubblica e una privata, OAUTH 2 utilizza un Bearer Token, inviato all'interno di un "Authorization" header. Solitamente lo standard più utilizzato ad oggi è quello del JWT.

3.4.3 Authorization Grant

Un altro punto fondamentale dell'OAUTH 2 è l'introduzione degli **Authorization Grant**, di cui i più utilizzati **Authorization Code**, **Client Credentials** e **Device Code**. Il tipo di authorization Grant tendenzialmente dipende da due fattori, cioè il metodo utilizzato dall'applicazione per richiedere l'autorizzazione e i tipi di grant supportati dall'API. A seconda del grant type utilizzato è possibile individuare grosse differenze nell'authentication flow, ed è pertanto opportuno scegliere quello corretto per la propria applicazione.

Grant Type: Authorization Code

Questo grant è tipicamente utilizzato per applicazioni **server-side**, dove il codice sorgente non viene esposto all'esterno permettendo quindi di poter mantenere la segretezza del **Client Secret**.
In breve, l'Authorization Code flow si svolge nel seguente modo:

1. L'utente ottiene un authorization code link contenente informazioni quali **client id**, **redirect uri**, **response type** e **scope**;

2. L'utente clicca sul link e deve effettuare il login al servizio per autenticare la propria identità;

3. Una volta autorizzata l'applicazione il servizio indirizza l'user-agent dell'applicazione al **redirect uri** specificato nel link, inserendo nel link **l'authorization code**;
4. L'applicazione utilizza l'authorization code per richiedere un access token;
5. L'applicazione riceve l'access token, che può ora utilizzare per effettuare richieste al servizio;

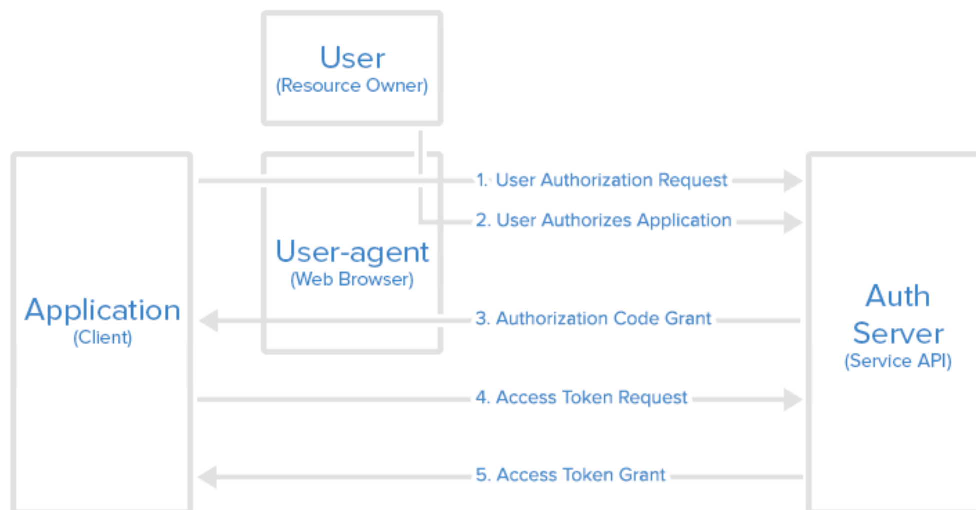


Figura 3.5: Authorization Code flow
Source: DigitalOcean.com

Grant Type: Client Credentials

Questo grant type è preferibile in situazioni dove non è possibile garantire la segretezza del client secret in quanto non è possibile proteggere il codice sorgente. Per questo motivo, a differenza dell'Authorization Code, nel Client Credentials flow l'utente ottiene l'access token immediatamente dopo aver inviato le proprie credenziali, rimuovendo interamente l'ottenimento di un authorization code.

Device Code Flow

Questo grant type è generalmente utilizzato per dispositivi che non sono in possesso di un browser o non sono in grado di utilizzare tutti gli input dei precedenti grant type per ottenere un access token. Si tratta quindi un grant type semplificato, utilizzabile ad esempio per applicazioni su dispositivi quali una SmartTV.
In questo caso, il flow è il seguente:

1. Il client invia una richiesta al **Security Token Service** includendo il proprio **client ID** e, opzionalmente, anche uno scope nel corpo della richiesta;
2. Il **Security Token Service** risponde con un **URL**, un **User Code** e un **Device Code**;

3. Il client espone questo URL e codice a schermo, via testuale o tramite un QR Code scannerizzabile, e si mette successivamente in attesa che il Security Token Service lo informi del risultato;

4. Mentre il client è in attesa, l'utente può utilizzare il link per effettuare l'autenticazione tramite un altro dispositivo;

5. Il Security Token Service invia la risposta al Client, includendo un Access Token;

6. L'utente può ora utilizzare tale Access Token per accedere alle API.

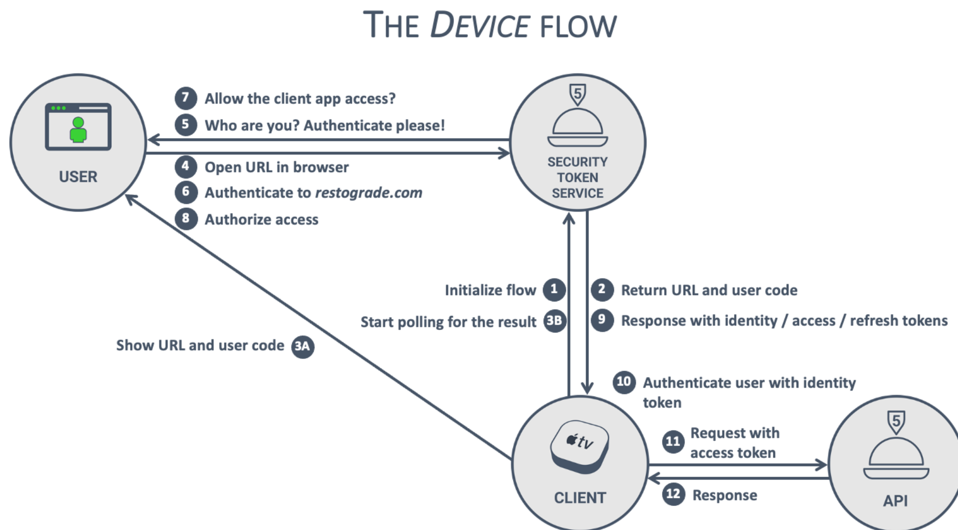


Figura 3.6: Device Flow
Source: PragmaticWebSecurity.com

3.5 OIDC

OpenID Connect è un protocollo complementare ad OAUTH 2, estendendone l'autenticazione con Single Sign-On. OIDC permette infatti di conservare e recuperare informazioni sugli end user e definisce gli scope OAUTH2 per permettere alle applicazioni di accedere a tali informazioni.

Per farlo OIDC definisce un ID token type da utilizzare in combinazione con gli access e refresh token di OAUTH 2, e anche in questo caso il provider di questo token è tendenzialmente l'authorization server. Nonostante si tratti di un protocollo complementare ad OAUTH 2, OIDC utilizza terminologie leggermente diverse, ma i concetti di fondo sono fondalmentalmente gli stessi:

- **End User**: colui che richiede l'accesso e le cui informazioni sono contenute nell'ID token;
- **OpenID Provider**: L'authorization Server che fornisce l'ID token;
- **Relying Party**: L'applicazione client che richiede il Token al provider;

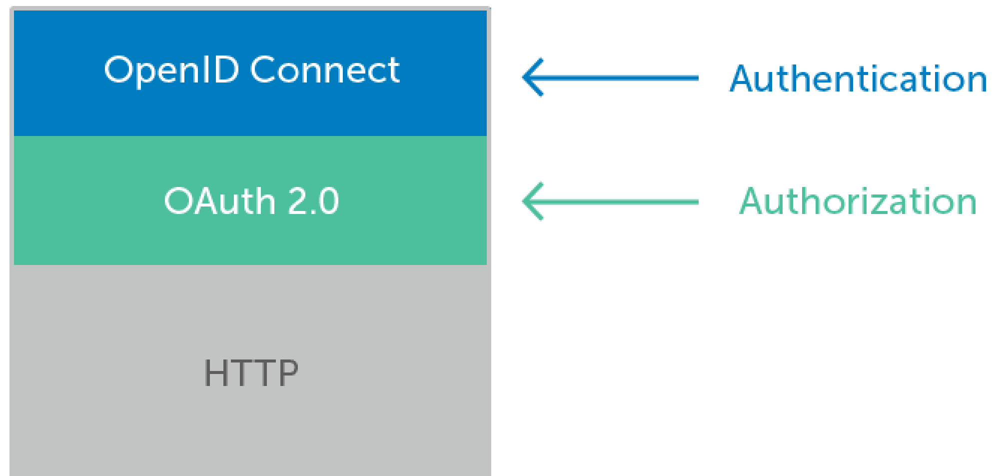


Figura 3.7: OpenID viene utilizzato in modo complementare ad OAUTH 2
Source: Okta.com

- **ID Token**: Il token fornito dall'OID Provider e che contiene le informazioni sull'utente;
- **Claims**: Informazioni sull'utente in forma di coppia nome-valore.

Capitolo 4

Sicurezza nel progetto

4.1 Da API individuali a gateway

L'idea proposta inizialmente dall'azienda era quella di realizzare API indipendenti, applicando la security su ciascuna di esse. Sarebbe, cioè, stato necessario replicare la definizione e gestione di Filter Chain, CORS, CSRF su ciascun microservizio, come anche l'operazione di verifica delle autorizzazioni. Questo, tuttavia, ci è sembrato estremamente dispendioso e ridondante: per ovviare a questo problema si è optato per una soluzione con l'utilizzo di un API Gateway, su cui sarebbe poi stata implementata la security. Poiché le API, una volta in produzione e quindi dockerizzate, sarebbero rimaste aperte solamente verso l'API Gateway, l'aggiunta di questo componente all'architettura ha consentito di centralizzare le operazioni di sicurezza, verificando l'identità e i permessi della richiesta prima di procedere all'indirizzamento verso i servizi.

4.2 Spring Cloud

Per costruire un gateway spring mette a disposizione una dipendenza molto semplice, quella di Spring Cloud, che permette di configurare un server Netty che agisca da Gateway, richiedendo solo che vengano configurate su di esso le Route (ovvero a quali servizi girare le richieste in arrivo al gateway) ed eventuali filtri, di cui parleremo più in basso.
Inoltre, poiché i Microservizi a cui il Gateway avrebbe fatto riferimento non sono stati ritenuti in numero eccessivo né è stato pianificato di deployarne multiple istanze, si è preferito - sotto richiesta dell'azienda - in questo stadio della progettazione saltare la fase di Service Discovery, mantenendo comunque la possibilità di includerla in caso in cui il progetto dovesse espandersi a tal punto da ritenerla necessaria.

```
return builder.routes()
        .route("userapi", r -> r.path("/UserApi/**")
                .filters(f -> f
                        .filter(responseInterceptor)
                        )
                .uri(httpUser))
        .route("logout", r -> r.path("/UserDetailsApi/logout/**")
                .filters(f -> f
                        .filter(jwtFilter)
                        .filter(responseInterceptor)
                        .filter(blacklistFilter))
                .uri(httpUser))
        .route("userdetails", r->r.path("/UserDetailsApi/**")
                .filters(f -> f
                        .filter(jwtFilter)
                        .filter(responseInterceptor))
                .uri(httpUser))
        .route("travels", r->r.path("/TravelApi/**")
                .filters(f -> f
                        .filter(jwtFilter)
                        .filter(responseInterceptor))
                .uri(httpTravel))
```

Figura 4.1: Un estratto delle routes del Gateway, configurate tramite Bean

4.3 JWT

Fin da subito, per autorizzare o negare le richieste, lo strumento scelto è stato quello del JSON Web Token; questo sistema è diventato un de facto standard per la comunicazione sicura tramite HTTP, ed è stato ritenuto valido anche per le necessità del nostro progetto. Un JWT si divide in **3 parti**, ossia:

- **Header**, ovvero la parte che contiene l'algoritmo di generazione del JWT;

- **Payload**, la parte che contiene le claims, dandoci informazioni sulla tipologia del token. Queste claims possono essere standard (ce ne sono 7 tipi, tra cui alcuni esempi quali "Issuer" e "Subject", ma è possibile inserire custom claims in base allo scopo che il JWT deve portare a termine: nel caso del progetto di stage, le claims a noi interessate erano l'userid e, eventualmente, il ruolo);

- **Signature**, una parte opzionale, ma utile per la sicurezza, calcolata tramite encoding dell'header e payload che vengono poi concatenate e ulteriormente criptate tramite l'algoritmo specificato nell'header.

Il Token viene poi criptato tramite il segreto, che non viene mai passato all'utente in modo da rendere impossibile a soggetti malintenzionati di modificare i dati inseriti nel Token.

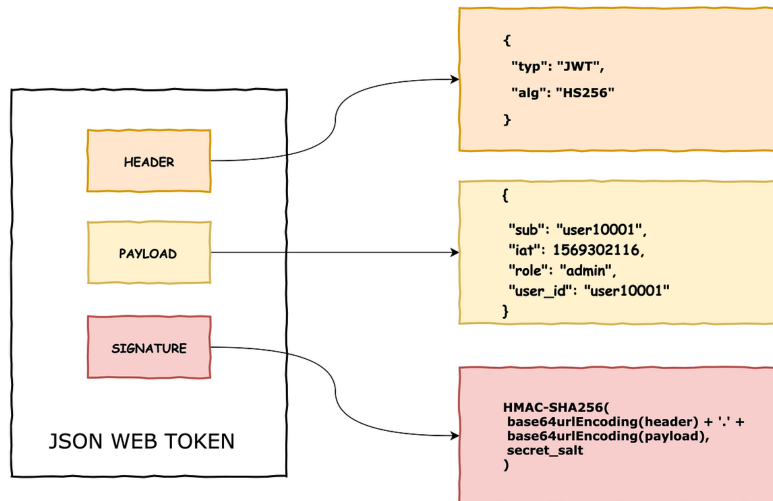


Figura 4.2: Un esempio del contenuto di un JWT
Source: Medium.com

4.3.1 Claims

Come menzionato precedentemente, le Claims sono informazioni riguardanti l'utente che vengono inserite all'interno del JWT tramite coppie nome-valore. In aggiunta ai 7 tipi standard è tuttavia possibile inserire all'interno del JWT delle Claims custom, variabili a seconda dello scopo del progetto. Nel nostro progetto, infatti, è sorta la necessità di avere diversi livelli di autorizzazione, in base a cui negare l'accesso ad un determinato microservizio. Per ovviare a questa necessità, è stato inserito un campo "ruolo" all'interno dell'entità utente nel database, che a sua volta verrà inserito nel corrispondente claim del JWT, in modo da poter verificare l'accesso a una determinata path.

4.3.2 Refresh Token

Se la parte precedente riguardava il token che viene utilizzato dall'utente per fare richiesta al gateway, ossia l'access token, questa sezione riguarda invece il Refresh Token, ossia quello che verrà utilizzato per ottenere un nuovo token alla scadenza del precedente.
Il Time to Live di ciascun access token, infatti, è generalmente molto breve (nel caso di questo progetto in particolare circa 20 minuti), mentre ai Refresh Token possono vivere molto più a lungo (in questo caso, una settimana): quando l'access token scade, l'utente prova a fare richiesta perché gli venga fatto un refresh inviando il proprio refresh token, ottenendo un nuovo access token in caso il refresh token sia ancora valido, o venendo rinviato al login in caso anche il Refresh Token sia scaduto, concedendo all'utente quindi di poter continuare a utilizzare i JWT finché il Refresh Token rimane valido.

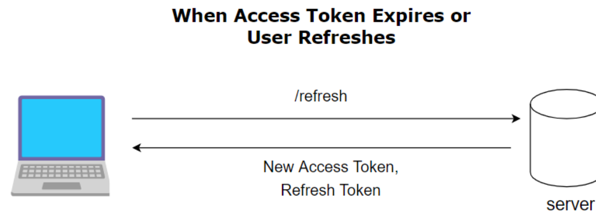


Figura 4.3: Richiesta di refresh
Source: GeeksForGeeks.com

4.4 Authentication Flow

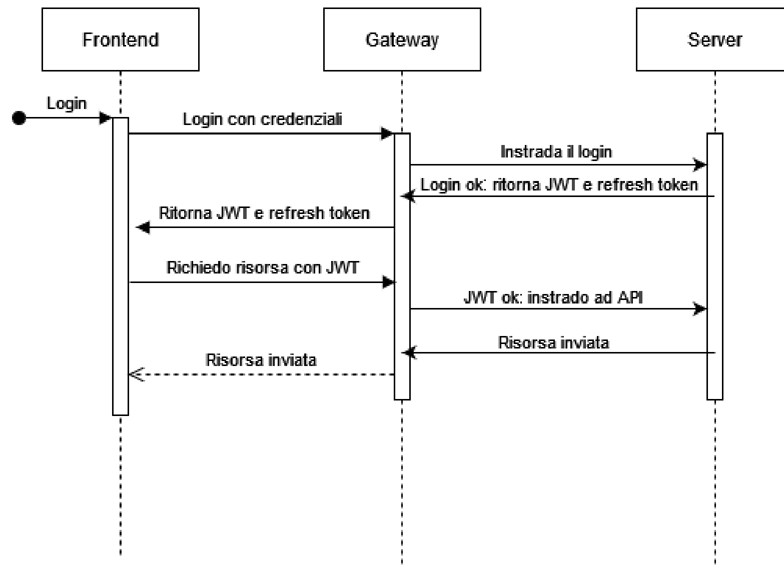


Figura 4.4: Esempio di Authentication Flow all'interno del progetto TripHippie

Il token viene generato nel momento del login: in caso l'utente effettui correttamente l'accesso tramite le proprie credenziali, viene generato un JWT che gli viene ritornato come risultato corretto della richiesta; l'utente dovrà a questo punto salvarlo e utilizzarlo per ogni richiesta che farà ai microservizi. Questo JWT Token sarà poi inviato al Gateway per l'accesso a risorse protette, e sarà l'unico modo per l'utente di comunicare ai servizi la sua identità. Il Gateway proverà successivamente a verificare che il JWT sia stato generato correttamente e non scaduto, e in caso di superamento dei controlli invierà la richiesta al microservizio, che procede a costruire un authentication (custom, in questo caso, per implementare il token nell'autenticazione) e la salva nel SecurityContext, da cui sarà poi possibile accedere all'interno di tutti i metodi relativi agli endpoint.

4.5 Blacklist

Nel corso di uno dei SAL settimanali è sorto tuttavia un altro problema: cosa fare dei token, ancora validi, al logout dell'utente? Questa problematica in particolare è sorta nel contesto in cui un utente, magari sospettoso che il proprio token sia stato intercettato, volesse fare in modo di negarne l'utilizzo. Per porre rimedio a ciò, al logout avvengono due cose:

- **Lato Back-End**: il refresh token viene eliminato, rendendo quindi impossibile ottenere un nuovo access token - che avrà vita estremamente breve senza effettuare nuovamente il login;

- **Lato Gateway**: un filtro osserva la risposta e - in caso di logout andato a buon fine - pone l'access token in una struttura chiamata ExpiringMap, che funziona da "blacklist": ogni volta che viene effettuata una richiesta - oltre ai normali controlli sul JWT - viene effettuato un ulteriore controllo, ossia la presenza dell'access token in questa blacklist, negando la richiesta in caso di risposta affermativa e rendendo quindi inutile l'access token anche durante la breve durata che gli resta; è stata scelta l'ExpiringMap per via della caratteristica peculiare di possedere un Time to Live per ogni entry, che viene rimossa al termine del suo ciclo vitale, in modo da non riempirla di token scaduti. Questa rimozione viene gestita autonomamente dalla struttura attraverso l'utilizzo di thread.

La possibilità di perdita dei dati in caso di crash del gateway è stata messa in considerazione, tuttavia poiché gli access token hanno già di loro vita breve (e ancora più breve sarà il tempo che passeranno nella blacklist in caso di logout), non è stato ritenuto un rischio particolarmente importante.

4.6 Intercettare e modificare il response body

Una richiesta per quanto riguarda la sicurezza era fare in modo che i microservizi potessero ritornare errori il più dettagliati possibile senza doversi preoccupare di star fornendo informazioni a potenziali attori malintenzionati: fondamentalmente, cioè, era necessario che questi errori giungessero al gateway, ma senza venire effettivamente instradati verso il cliente, che avrebbe dovuto vedere soltanto un generico messaggio d'errore. Per implementare questa richiesta mi sono affidato all'utilizzo di un postfiltro globale, ovvero un filtro che sarebbe entrato in azione solamente al ricevimento della risposta da parte del servizio, con della logica al suo interno per analizzare lo stato e sostituire il corpo della risposta con un generico messaggio d'errore relativo a quell'HTTPStatus (ad esempio, un errore 404 restituirebbe sempre all'interno del proprio corpo "The resource was not found", indipendentemente dal livello di dettaglio ritornato dal servizio, che viene semplicemente loggato invece che ritornato).

4.7 CORS

Il Cross Origin Resource Sharing è un sistema basato su header HTTP che permette al server di indicare a quali origini permetterà di accedere alle proprie risorse; se non configurato, il server negherà l'accesso a qualunque richiesta non provenga da se stesso (questo ovviamente non vale per development tool quali ad esempio Postman). Le intestazioni CORS possono essere di diverso tipo, tra le quali:

- **Access-Control-Allow-Origin**: Definisce le origini consentite;
- **Access-Control-Allow-Credentials**: Definisce se consentire o meno le richieste nonostante la modalità credenziali sia impostata su include;
- **Access-Control-Allow-Headers**: Definisce le intestazioni utilizzabili;
- **Access-Control-Allow-Methods**: Definisce i metodi HTTP consentiti.

Il metodo più semplice per configurarlo in Spring Gateway è quello di farlo in un file di risorse .yaml (o .yml), che verrà poi letto e implementato al deploy in modo simile a come verrebbe fatto con un oggetto **Bean**. Poiché effettuiamo già controlli su ogni richiesta in arrivo tramite un sistema JWT, è stato scelto di mantenere aperte richieste da ogni origine, sapendo che attori malintenzionati verrebbero comunque bloccati in fase di verifica del JWT.

4.8 Spring Security

Nonostante l'implementazione di controlli di sicurezza basati sui JWT all'interno del gateway, ho comunque avuto bisogno di configurare **Spring Security** all'interno dei diversi microservizi per rispondere a una problematica sorta durante lo sviluppo dei servizi, ossia quella di ottenere l'utente autenticato senza far sì che sia lui stesso a comunicarlo in qualche modo che non sia tramite JWT. Un'altra possibilità infatti sarebbe stata quella di ottenere il token dall'header ogni volta ed estrarne i dati all'interno dell'endpoint, tuttavia questo sarebbe risultato un metodo dalla scalabilità estremamente bassa, anche perché mi avrebbe forzato a implementare la logica dietro alla validazione e ottenimento dei dati da un JWT all'interno di ogni servizio. Fortunatamente, Spring Security mette a disposizione uno strumento fondamentale, cioè il **SecurityContextHolder**.

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {

    http
            .addFilterBefore(authenticationFilter, BasicAuthenticationFilter.class)
            .authorizeHttpRequests((auth) -> auth
                    .requestMatchers("/UserApi/**").permitAll()
                    .requestMatchers("/UserDetailsApi/**").authenticated()
                    )
            .csrf(csrf -> csrf.disable())
            .httpBasic(Customizer.withDefaults());
    return http.build();

}
```

Figura 4.5: Configurazione della filter chain all'interno di UserService

4.8.1 Security Context Holder

Il Security Context Holder, come dice il nome, possiede il contesto, ed è richiamabile in qualunque endpoint. Al suo interno è possibile salvare un **Authentication** che a

sua volta possiede un **Principal**, ossia l'utente loggato vero e proprio. Nonostante Spring metta a disposizione implementazioni standard di queste classi, ho dovuto crearne alcune di customizzate per l'occasione, per il semplice motivo che la mia autenticazione non si sarebbe basata su credenziali come password, che non viene mai passata all'utente, quanto invece sul token e sui dati che inserisco al suo interno, come id e ruoli. Chiamando un filtro, quindi, vado a costruire una mia implementazione, chiamata **TokenBasedAuthentication**, e la salvo all'interno del Security Context Holder, che ogni endpoint potrà acquisire in modo molto semplice, potendo quindi sapere in base al token quale utente sta compiendo la richiesta.

```
@PutMapping("/modifyEmail")
public ResponseEntity<Object> modifyEmail(
        TokenBasedAuthentication authentication,
        @RequestBody ModifyEmailDto modifyEmailDto) {
```

Figura 4.6: E' sufficiente inserire l'authentication negli input della funzione per costruirla a partire dal contesto

4.8.2 CSRF

Il Cross-site Request Forgery è un tipo di attacco che sfrutta la fiducia di un sito nel browser dell'utente, che può essere costretto inconsapevolmente a inviare alcune richieste - che un hacker potrebbe nascondere in elementi HTML o URL - ai servizi.

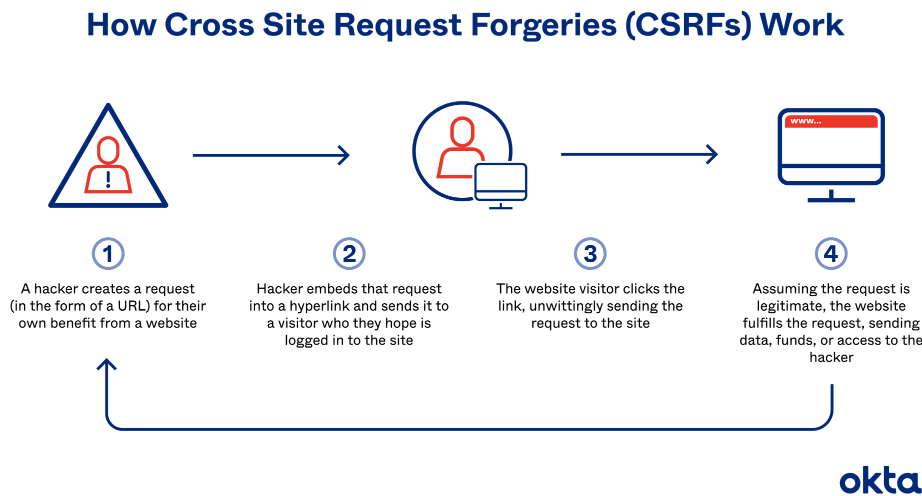


Figura 4.7: Funzionamento di un attacco CSRF
Source: Okta.com

Di norma, poiché chi attacca non riceve la risposta, per portare a termine un attacco CSRF si utilizzano richieste che comportano un cambio di stato, quali ad esempio la

cancellazione di un record, la modifica di una password, l'acquisto di un prodotto e così via. Per questo motivo, di default Spring Security blocca tutte le richieste ad endpoint che non siano di tipo GET, anche quelli configurati come **"permit all"**, mentre gli endpoint GET rimangono liberamente accessibili. Come si può vedere sopra, all'interno del progetto è disabilitato; questo perché il progetto implementa già JWT, uno standard di sicurezza superiore a quello del CSRF token, e gli unici endpoint che vengono lasciati esposti sono considerati non pericolosi (ad esempio, il login o la registrazione).

Capitolo 5

Implementazione OAUTH2 tramite Keycloak

5.1 Introduzione

Avendo completato i compiti a me assegnati prima del tempo, ho domandato al mio tutor aziendale il permesso di esplorare la possibilità di utilizzare il protocollo OAuth 2 all'interno del nostro progetto durante l'ultima settimana ottenendo risposta affermativa. Avendo già studiato il protocollo in una fase precedente avevo già chiaro il suo funzionamento, tuttavia non ero sicuro di come realizzare l'authorization server vero e proprio.

5.2 Scelta dell'Authorization Server

La prima scelta per l'implementazione di OAUTH2 è stata, ovviamente, quella dell'authorization server: in questo caso ero alla ricerca di qualcosa su cui poter mantenere un minimo di controllo (in caso di voler, ad esempio, customizzare la registrazione o il login con altri campi) e che fosse, almeno in questa fase dello stage, gratuito.
Per questi motivi la mia scelta è caduta su **Keycloak**, un software open-source per consentire il **Single Sign-On** con implementazione di verifica dell'identità tramite JWT. Keycloak si basa su protocolli standard e offre supporto per **OAuth 2**, **OpenID Connect** e **SAML**, e consente di mantenere un certo livello di controllo sulla base di dati configurandolo, ad esempio, verso un database postgres (è possibile inoltre integrarlo tramite docker), rispondendo quindi perfettamente ai requisiti che cercavo. Keycloak offre inoltre altre funzionalità come ad esempio il social login tramite piattaforme quali **Google**, **Github** o **Facebook**.

Figura 5.1: Logo Keycloak

5.3 Struttura Keycloak

Keycloak si basa sull'utilizzo di **Realms**, ovvero degli ambienti completamente indipendenti tra di loro dove è possibile configurare i propri client, user, ruoli e scope. Una volta creato il proprio realm, si procede a configurare tutto il necessario al suo interno: in particolare, la prima cosa che ho definito sono stati i **Client** e il loro authentication flow, poi gli utenti (customizzabili tramite attributi) e i ruoli. Poichè il mio client sarebbe stato una pagina angular e, quindi, non un posto sicuro per memorizzare il client secret, ho optato per utilizzare il grant type **Client Credentials**, cioè utilizzando solo le credenziali utente per l'ottenimento dell'access token.

5.4 Integrazione all'interno del progetto

Il passaggio successivo è stato quindi quello di integrare il SSO di Keycloak con il resto dell'applicazione: come prima cosa ho rimosso login e registrazione dal back-end, in quanto ora compiti gestiti direttamente da Keycloak, come anche tutti gli endpoint che andavano a modificare le informazioni personali degli utenti; in concondanza ho modificato il routing del gateway in quanto questi servizi non risultavano più offerti dal nostro back-end. Le API, diventando ora resource server, riferiscono a keycloak come **Introspection URI** a cui fare riferimento per la validazione del token. Nel front-end, invece, ho innanzitutto installato la libreria di Keycloak, sostituito Guard con l'implementazione standard fornite da keycloak e rimosso l'interceptor, in quanto la libreria inserisce automaticamente un bearer token all'interno di ogni richiesta http uscente; infine, ho modificato login e registrazione implementando le funzioni standard di Keycloak.

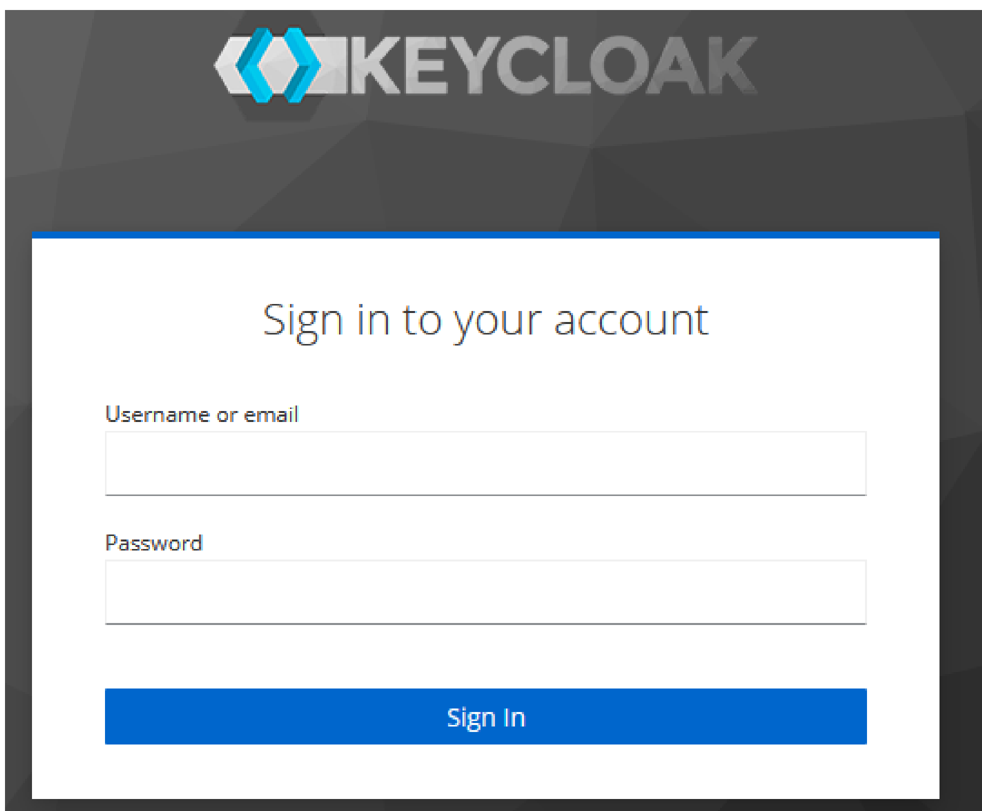


Figura 5.2: SSO tramite Keycloak

5.5 Considerazioni finali

Come accennato all'inizio del capitolo, il tempo dedicato all'implementazione di questo protocollo nel progetto è stato molto limitato e più a scopo esplorativo che di avere un prodotto finale funzionante; nonostante l'integrazione dell'autenticazione fosse funzionante, infatti, il tempo non è stato sufficiente per implementare tutte le modifiche ai servizi e al front-end necessarie per una corretta integrazione con Keycloak, mantenendo questa versione come un "Work in Progress" da eventualmente terminare in futuro; non essendo questa funzionalità parte del piano di lavoro, tuttavia, questo non è andato a inficiare sul raggiungimento degli obiettivi previsti.

Capitolo 6

Implementazioni sul front-end

6.1 Introduzione

Nonostante il grosso del mio lavoro abbia riguardato il gateway e la messa in sicurezza del progetto, mi è stato anche assegnato il compito di fare alcune implementazioni lato front-end per integrare correttamente il front-end con i meccanismi di security del progetto. Per fare ciò, ho fatto alcune semplici implementazioni, ponendo la logica principalmente in due componenti che angular mette a disposizione dello sviluppatore, ossia tramite un **httpinterceptor** e un **guard**.

6.2 HttpInterceptor

Come suggerisce il nome, un HttpInterceptor è un componente che si occupa di "intercettare" richieste http ed effettuare determinate operazioni su di essa prima di spedirla verso la sua destinazione. Questo concetto permette all'applicazione angular di essere estremamente scalabile - permettendo di centralizzare questa logica invece di doverla riprodurre su ogni richiesta individualmente - ed è perfetto per implementare la comunicazione utilizzando JWT. In particolare, in questo progetto l'HttpInterceptor

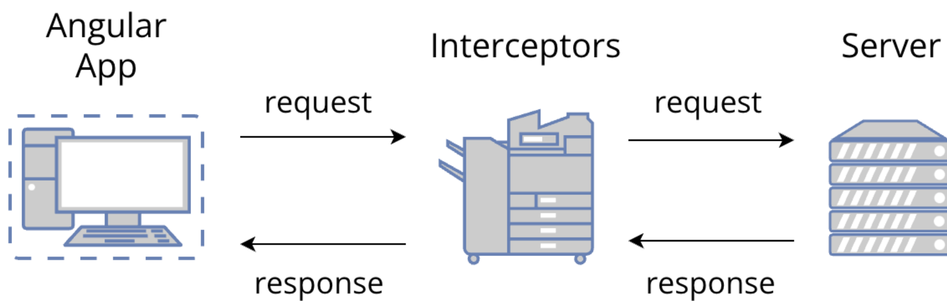


Figura 6.1: Il flow di una http request in Angular con l'utilizzo di un HttpInterceptor
Source: Dev-Academy.com

che ho scritto ha due compiti: quello di aggiungere **l'Authorization** header contenente il token a ciascuna richiesta che deve accedere a una risorsa privata, e quello di provare a effettuare una richiesta di refresh dell'access token nel caso in cui tale chiamata

fallisca con errore 401 (Unauthorized) per poi riprovare nuovamente (in caso di ulteriore fallimento verrebbe chiamato il logout, in quanto significherebbe che anche il refresh token è scaduto).

6.3 Guard

Nonostante non sia in grado di accedere a risorse private in assenza di credenziali, non sarebbe comunque buona pratica mostrare a un utente pagine private - spesso bianche o con errori dovuti dal mancato caricamento dei dati - in assenza di una corretta autenticazione (ad esempio accedendo direttamente tramite link). Per impedire che questo accada Angular mette a disposizione un componente, il Guard, che viene applicato alle pagine all'interno del routing module dell'applicazione. Al suo interno possono essere definite operazioni - ad esempio, si può controllare che l'utente sia in possesso di token o altri dati - prima di permettergli l'accesso alla pagina, e reindirizzarlo altrove (nel nostro caso alla pagina di login) in caso ne sia sprovvisto.

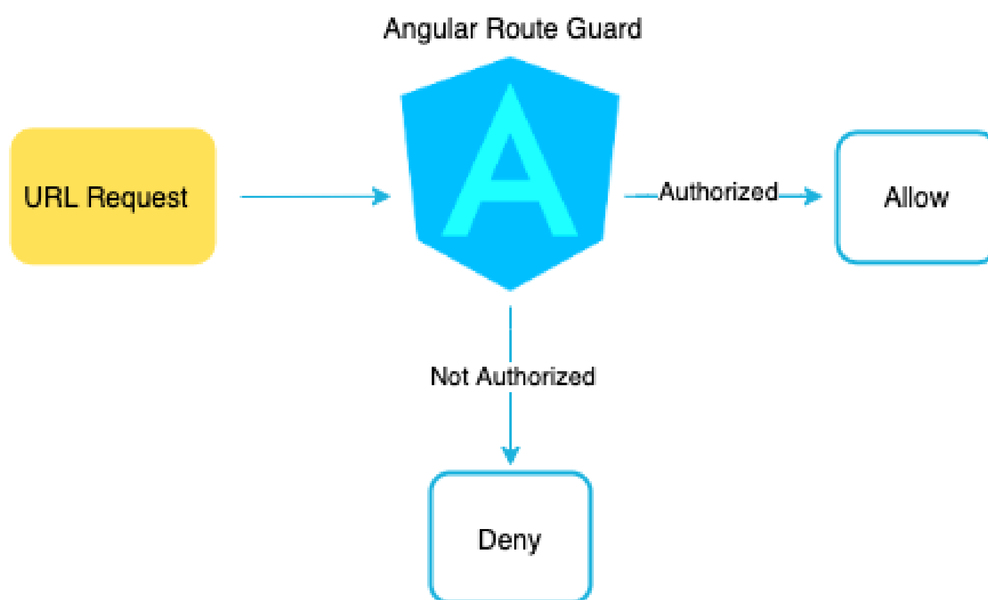


Figura 6.2: Il funzionamento di un Guard in breve
Source: medium.com

Capitolo 7

Verifica e validazione

7.1 Introduzione

Sebbene non parte degli obiettivi o delle richieste aziendali, il codice che ho scritto è stato rigorosamente testato per verificarne il corretto funzionamento. I problemi principali sono stati riscontrati durante il testing dei filtri del gateway, problemi che purtroppo non sono riuscito a risolvere interamente nel corso dello stage.

7.2 Strumenti utilizzati

Sono stati utilizzati principalmente 3 strumenti per verificare il corretto funzionamento del codice:

- **Postman**: Un strumento per semplificare lo sviluppo di API permettendo di simulare facilmente chiamate http in un ambiente controllato;
- **Junit**: Il framework ormai considerato standard per effettuare test in Java;
- **Mockito**: Uno dei framework per mocking Java più utilizzati al mondo, incredibilmente utile per simulare il comportamento di altri membri di una classe in modo da semplificare e controllare al meglio lo svolgimento degli unit test. Fondamentale sopratutto per effettuare il mocking delle chiamate http nei filtri.

7.3 Code coverage conseguita

7.3.1 Gateway

I test sul gateway si sono concentrati sui filtri e sui servizi da essi utilizzati.

src/main/java	73,2 %	624	228	852
it.Triphippie	37,5 %	3	5	8
it.Triphippie.config	100,0 %	218	0	218
it.Triphippie.Exceptions	50,0 %	4	4	8
it.Triphippie.filter	45,3 %	169	204	373
it.Triphippie.services	93,9 %	230	15	245

Figura 7.1: Code Coverage package security

Sfortunatamente per via di alcuni problemi nell'unit testing dei filtri la code coverage raggiunta è stata solo del circa 73%.

7.3.2 Security servizi

Per quanto riguarda la security sui microservizi, sono state testate tutte le classi create per l'implementazione raggiungendo un Code Coverage del 100%.

it.synclab.triphippie.security	100,0 %	239	0	239
AnonAuthentication.java	100,0 %	30	0	30
CustomUserDetailsService.jav	100,0 %	23	0	23
RefreshTokenService.java	100,0 %	53	0	53
SecurityUser.java	100,0 %	42	0	42
TokenBasedAuthentication.ja	100,0 %	23	0	23
TokenGenerationService.java	100,0 %	43	0	43
TokenUtils.java	100,0 %	25	0	25

Figura 7.2: Code coverage package security

it.synclab.triphippie.security	100,0 %	239	0	239
AnonAuthentication.java	100,0 %	30	0	30
CustomUserDetailsService.jav	100,0 %	23	0	23
RefreshTokenService.java	100,0 %	53	0	53
SecurityUser.java	100,0 %	42	0	42
TokenBasedAuthentication.ja	100,0 %	23	0	23
TokenGenerationService.java	100,0 %	43	0	43
TokenUtils.java	100,0 %	25	0	25

Figura 7.3: Code coverage package filter

7.4 Collaudi e validazione

Durante l'ultima settimana di stage si è svolto il collaudo provando il corretto funzionamento dell'applicazione nelle sue diverse componenti: esso si è tenuto su Google Meet durante un SAL con in presenza tutti i programmatori sul progetto, il mio tutor Fabio Pallaro e un altro ingegnere dell'azienda, Daniele Zorzi, nostre figure di riferimento durante lo svolgimento del progetto. Entrambi si sono detti soddisfatti dei risultati raggiunti rendendo quindi questa attività un successo rispetto agli obiettivi prefissati.

Capitolo 8

Conclusioni

8.1 Obiettivi Raggiunti

Lo stage si è svolto interamente entro i tempi e le modalità previste e tutti gli obiettivi sono stati raggiunti, inclusi gli obiettivi desiderabili e quelli opzionali.

8.2 Conoscenze acquisite

Nel corso dei 2 mesi di stage ho potuto approfondire una moltitudine di tematiche professionali, non solo dal punto di vista dei protocolli di sicurezza ma anche arricchendo le mie conoscenze dal punto di vista dei linguaggi di programmazione, framework utilizzati e progettazione nonché espandendo la mia capacità di lavorare in autonomia. Di seguito illustrerò più nel dettaglio gli apprendimenti derivati da questa attività.

8.2.1 Protocolli di sicurezza

Il primo e più ovvio è l'aspetto su cui mi sono concentrato, ossia quello dei protocolli di sicurezza. Seppur avendo già una basilare conoscenza di come funzioni la comunicazione sul web, ero fondamentalmente ignaro dei processi specifici che venivano utilizzati per mettere in sicurezza i dati personali di un utente durante le comunicazioni; durante gli scorsi mesi ho potuto impararne il funzionamento non solo a livello prettamente teorico ma approfondendo e testando con mano i vari aspetti della sicurezza, capendone anche i punti deboli e dove prestare più attenzione.

8.2.2 Progettazione

Per quanto riguarda le mie competenze progettuali, sopratutto nella prima fase del progetto - lo studio individuale - ho potuto approfondire diversi design pattern utilizzati per la comunicazione, come ad esempio quello del **Circuit Breaker** o il **Saga Pattern**, mentre nella seconda parte del progetto ho acquisito conoscenza maggiore principalmente sulle difficoltà implementative derivanti dalle scelte fatte in precedenza e come lavorarci attorno.

8.2.3 Way of Working

Nel corso dell'attività didattica ho potuto toccare con mano quello che è un vero e proprio Way of Working aziendale fondato sul metodo **agile**, migliorando la mia capacità di lavorare all'interno di un team e di implementare in modo corretto alcune pratiche quali quella della **Continuous Integration**. Ho inoltre potuto utilizzare strumenti nuovi per la coordinazione di attività, quali ad esempio Trello, di cui non ero precedentemente a conoscenza.

8.3 Evoluzione rispetto a consuntivo

Il progetto è proseguito più velocemente di quanto previsto nel consuntivo, ponendomi infatti nella posizione di sfruttare la mia ultima settimana per esplorare qualcosa di non previsto nel piano di lavoro in ottica migliorativa; in particolare, mi sono trovato in anticipo sulla tabella di marcia nella prima fase, quella dello studio individuale, grazie alla mia conoscenza pregressa del linguaggio Java e di alcuni dei framework utilizzati.

8.4 Valutazione personale

L'attività di stage si è rivelata immensamente utile non solo per permettermi di acquisire un solido bagaglio di conoscenze teoriche ma anche e sopratutto per aiutarmi a comprendere lo svolgimento pratico di un progetto in team. Dopo un progetto di Ingegneria del Software che si è rivelato ben più turbolento di quanto avessi potuto immaginare, infatti, è stato quasi rigenerante venire inserito all'interno di un progetto funzionante e trovare attorno a me figure disponibili e capaci. Questa esperienza mi ha permesso di crescere sia dal punto di vista personale che da quello professionale, ponendomi diverse sfide che mi hanno aiutato a maturare. Nella prima fase del progetto ho potuto comprendere con mano cosa significhi apprendere a programmare al di fuori di un contesto prettamente scolastico, organizzando il mio apprendimento e leggendo fonti che non ero solito consultare, quali ad esempio documentazione di framework. Ho potuto inoltre capire le difficoltà relative a compiere decisioni progettuali seguendo solo delle linee guida, senza essere tenuto per mano da un docente che mi dicesse cosa fare.

La mia valutazione finale sullo stage, dunque, non può che essere nettamente positiva, e vorrei chiudere il documento ribadendo l'importanza che ritengo possa avere per uno studente un'esperienza di questo tipo, specie nel contesto di un panorama come quello dell'istruzione italiana che, a mio modesto parere, pone spesso troppa enfasi sull'aspetto teorico che non sempre risulta essere adeguatamente bilanciato da prove pratiche.

Acronimi e abbreviazioni

API Application Programming Interface. 4, 8, 15, 17, 37

CORS Cross Origin Resource Sharing. 17, 21, 38

CRUD Create Read Update Delete. 2

CSRF Cross-site Request Forgery. ix, 17, 23, 38

CSS Cascading Style Sheets. 2

HTML HyperText Markup Language. 2, 23

HTTP HyperText Transfer Protocol. ix, 9–11, 21, 22, 38

HTTPS HyperText Transfer Protocol . ix, 9–11, 39

IDE Integrated Development Environment. 4, 39

IOT Internet of Things. 1, 39

IT Information Technology. 1

JPA Java Persistence API. 2, 4, 39

JWT JSON Web Token. iii, ix, 4, 13, 18, 19, 22, 39

OAUTH Open Authorization . ix, 11–13, 15, 16, 39

OIDC OpenID Connect. 15, 39

ORM Object Relational Mapping. 4, 40

REST REpresentational State Transfer. 4, 40

SAL Stato Avanzamento Lavori. 1

SOLID Single responsibility principle, Open/closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle. 4

SSO Single Sign-On. ix, 27, 41

TLS Transport Layer Security. 10, 41

TTL Time to Live. 21, 41

Glossario

.yaml File di configurazione.. 22

Access Token Token di breve durata che permette all'utente di accedere a risorse protette senza esporre le proprie credenziali.. 15

Agile Le metodologie di sviluppo software agile consistono nel rilasciare rapidamente modifiche al software in piccole porzioni con l'obiettivo di migliorare la soddisfazione dei clienti.. 1

AngularJS Framework JavaScript per applicazioni web dinamiche, utilizzato in particolare per la creazione di SPA e web app.. 2

API Un intermediario software grazie al quale due applicazioni possono comunicare tra loro.. 35

Authorization Server In OAuth 2, la figura che si occupa di verificare l'identità dell'utente e che effettua il rilascio degli access token allo stesso per poter accedere all'applicazione.. 12, 13

Back-End E' la parte non visivibile all'utente che approda sulla piattaforma. 2, 3

Bean Un'istanza di una classe Java che viene gestita dal container Spring.. ix, 18, 22

Bearer Token Un token inviato all'interno dell'header HTTP "Authorization".. 13

Big Data Una raccolta di dati informatici così estesa in termini di volume, velocità e varietà da richiedere tecnologie e metodi analitici specifici per l'estrazione di valore o conoscenza.. 1

Cascading Style Sheets Linguaggio usato per definire la formattazione di documenti HTML, XHTML e XML, ad esempio i siti web e relative pagine web. . 35

Claims Informazioni inserite in un token in formato coppia nome-valore.. 19

Client Una componente che accede ai servizi o alle risorse di un'altra componente, detta server. 2, 15

Cloud Computing Tecnologia che consente di usufruire, tramite server remoto, di risorse software e hardware (come memorie di massa per l'archiviazione di dati), il cui utilizzo è offerto come servizio da un provider.. 1

Code Coverage La percentuale di codice attraversato dei test rispetto al totale della code base. ix, 31, 32

CORS Caratteristica di sicurezza del browser che limita le richieste HTTP multiorigine avviate da script in esecuzione nel browser.. 21, 35

CSRF Metodo d'attacco utilizzato prevalentemente per le truffe via Internet dove i criminali informatici riprendono una sessione autorizzata dall'utente, riuscendo così a eseguire azioni dannose.. 23, 35

Database Archivio di dati strutturato in modo da razionalizzare la gestione e l'aggiornamento delle informazioni e da permettere lo svolgimento di ricerche complesse.. 7

Dependency Injection Design pattern utilizzato nella programmazione ad oggetti, utile per semplificare la scrittura del codice e la testabilità dei programmi.. 2

Discord Piattaforma di instant messaging, utilizzata molto frequentemente per chiedere e/o ottenere informazioni veloci e coordinarsi per le giornate in presenza.. 2

DNS Hijacking Un attacco in cui gli indirizzi DNS vengono dirottati e il traffico reindirizzato verso falsi server DNS.. 11

Domain Spoofing Un tipo di attacco informatico cha va a modificare le associazioni tra nomi a dominio e indirizzi IP su un server DNS per reindirizzare i dati verso server controllati dall'attaccante.. 11

ExpiringMap Una classe che estende Map con degli oggetti che si rimuovono automaticamente una volta scaduto il time to live tramite l'utilizzo di thread.. 21

Filter Chain Un oggetto fornito allo sviluppatore per dare una visione alla catena d'invocazione di una richiesta con filtri per una risorsa.. 17

Front-End La parte visibile all'utente di un programma e con cui egli può interagire.. 3

Gateway Un dispositivo di rete che collega due reti informatiche di tipo diverso.. ix, 4, 17, 18

Google Meet Un'applicazione di video conferenza sviluppata da Google.. 1, 2

Guard Componente di angular che permette di eseguire della logica di verifica prima di permettere (o negare) l'accesso a una pagina web.. 30

Header In HTTP, direttive, relative alla sicurezza del sito web, che vengono trasmesse attraverso l'HTTP header response. In JWT, contiene informazioni sull'algoritmo di generazione.. 18

HTTP Un protocollo a livello applicativo usato come principale sistema per la trasmissione d'informazioni sul web ovvero in un'architettura tipica client-server.. 35

HttpInterceptor Componente di Angular che intercetta le richieste http uscenti per poter eseguire della logica su di esse.. 29

HTTPS Un protocollo per la comunicazione sicura attraverso una rete di computer utilizzato su Internet, evoluzione di HTTP.. 35

HyperText Markup Language Linguaggio descrittore delle pagine web.. 35

IDE Una suite software che racchiude in un'unica interfaccia utente grafica i principali strumenti di sviluppo per codificare software. 35

IOT La rete di oggetti fisici che hanno sensori, software e altre tecnologie integrate allo scopo di connettere e scambiare dati con altri dispositivi e sistemi su Internet.. 35

Java Linguaggio di programmazione che permette di sviluppare programmi eseguibili su diversi tipi di computer e compatibili con qualsiasi sistema operativo.. 2

JPA Un framework per il linguaggio di programmazione Java che si occupa della gestione della persistenza dei dati di un DBMS relazionale nelle applicazioni che usano le piattaforme Java Platform, Standard Edition e Java Enterprise Edition.. 35

Javascript Linguaggio di programmazione orientato agli eventi utilizzato sia lato client web sia server.. 2

Junit Un framework di unit testing per il linguaggio di programmazione Java.. 31

JWT Un sistema di cifratura e di contatto in formato JSON per lo scambio di informazioni tra i vari servizi di un server.. 35

Keycloak Prodotto software open source per consentire il single sign-on con gestione dell'identità e degli accessi mirato ad applicazioni e servizi moderni.. 26

Man in the Middle Un attacco informatico in cui qualcuno segretamente ritrasmette o altera la comunicazione tra due parti che credono di comunicare direttamente tra di loro.. 10

Microservizi Architettura composta da servizi indipendenti tra loro.. 4, 17

Mockito Framework java Mocking che mira a fornire la capacità di scrivere un test unitario leggibile utilizzando la sua semplice API.. 31

Monolite Architettura dove tutte le funzioni all'interno di un unico blocco. 4

Netty Framework client-server per lo sviluppo di applicazioni Java per le comunicazioni telematiche, come server e client di protocollo.. 17

OAUTH Un metodo sicuro per consentire agli utenti di concedere ai fornitori di servizi (ad esempio, siti web e applicazioni) l'accesso alle proprie informazioni senza fornire loro le password.. 35

OIDC Un protocollo che estende il protocollo di autorizzazione OAuth 2.0 da usare come protocollo di autenticazione aggiuntivo.. 15, 35

On-Path Attacks Un sinonimo di Man in the Middle Attack.. 11

ORM una tecnica utilizzata per creare un "ponte" tra programmi orientati agli oggetti e, nella maggior parte dei casi, database relazionali.. 35

Payload Il corpo di un messaggio.. 18

Postman Piattaforma API per costruire e usare API.. 21, 31

Refresh Token Token memorizzato e consegnato all'utente in modo che possa ottenere un nuovo access token quando quello in suo possesso scade.. 3, 19

Resource Owner In OAuth 2, l'utente che autorizza l'applicazione ad accedere alle proprie risorse.. 12

Resource Server In OAuth 2, la figura che si occupa di gestire richieste autenticate dopo che l'utente ha ottenuto un access token dall'authorization server.. 12, 13

REST Un sistema di trasmissione di dati su HTTP senza ulteriori livelli.. 35

Route L'indirizzo a cui effettuare il reinderizzamento per una data richiesta.. 17

Scope Descrive i permessi dell'utente, tipicamente associato ad un token.. 13

Security Token Service Un servizio che consente di richiedere credenziali temporanee con privilegi limitati per gli utenti.. 14, 15

SecurityContext Classe utilizzata per conservare i dati dell'utente attualmente autenticato per una richiesta.. 20

SecurityContextHolder Classe che contiene il contesto corrente.. 22

Server Un programma che offre un servizio specifico e che può essere richiesto localmente o all'interno di una rete da altri programmi. 2

Service Discovery Il processo di individuamento delle API in un'architettura a microservizi.. 17

Signature Componente di un JWT opzionale ma utile per la sicurezza.. 18

Smart Working Una modalità di esecuzione del rapporto di lavoro subordinato caratterizzato dall'assenza di vincoli orari o spaziali.. 1

Software House L'insieme delle azioni volte a difendere computer, server, dispositivi mobili, sistemi elettronici, reti e dati dagli attacchi dannosi.. 1

Software House Azienda che si occupa dell'elaborazione e della commercializzazione di programmi per elaboratori.. 1

Spring Un framework open source per lo sviluppo di applicazioni su piattaforma Java.. 2

Spring Boot Uno strumento che semplifica e velocizza lo sviluppo di applicazioni web e microservizi con Spring Framework. 2, 4

Spring Cloud Soluzione di convenzione sulla configurazione di Spring per la creazione di applicazioni Spring di livello di produzione con quantità minime di configurazione.. 17

Spring Data Uno strumento che permette di semplificare lo stato di persistenza rimuovendo completamente l'implementazione dei DAO dalla nostra applicazione.. 2, 4

Spring Security Un framework Java/Java EE che fornisce autenticazione, autorizzazione e altre funzionalità di sicurezza per le applicazioni aziendali.. iii, 2

SSO La proprietà di un sistema di controllo d'accesso che consente ad un utente di effettuare un'unica autenticazione valida per più sistemi software o risorse informatiche alle quali è abilitato.. 15, 35

StopLight Piattaforma di progettazione e sviluppo di API che offre strumenti per progettare, testare e documentare le API.. 4

TLS Evoluzione più sicura di SSL.. 35

Token Un oggetto digitale che contiene informazioni sull'identità dell'entità che effettua la richiesta e sul tipo di accesso per cui è autorizzato.. 13, 18

Trello Strumento di gestione di progetti, permette la creazione e gestione semplificata di bacheche per mantenere le task organizzate.. 2

TTL Tempo di vita di un oggetto.. 19, 35

Web Application Una applicazione web (web application in inglese, abbreviato web app), in informatica ed in particolare nella programmazione web, indica genericamente tutte le applicazioni distribuite ovvero applicazioni accessibili/fruibili via web per mezzo di un network.. iii

Bibliografia

Siti web consultati

An Introduction to Oauth 2. URL: https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2.

Attacchi CSRF. URL: https://kinsta.com/it/blog/attacchi-csrf/.

Cross-site Request Forgery. URL: https://it.wikipedia.org/wiki/Cross-site_request_forgery.

Difference between SSL and TLS. URL: https://aws.amazon.com/it/compare/the-difference-between-ssl-and-tls/.

Everybody Wins with the Device Flow. URL: https://pragmaticwebsecurity.com/articles/oauthoidc/device-flow.html.

Introduction to OAuth. URL: https://oauth.net/about/introduction/.

JWT Authentication with refresh tokens. URL: https://www.geeksforgeeks.org/jwt-authentication-with-refresh-tokens/.

Manifesto Agile. URL: http://agilemanifesto.org/iso/it/.

Mockito. URL: https://site.mockito.org/.

What is OAuth. URL: https://developer.okta.com/blog/2017/06/21/what-the-heck-is-oauth.

Why HTTP is not secure. URL: https://www.cloudflare.com/learning/ssl/why-is-http-not-secure/.