REALIZZAZIONE DI UNA APPLICAZIONE HOME BANKING CON ARCHITETTURA RESTFUL IBRIDATA

Università degli Studi di Firenze
Corso di Laurea Magistrale in Ingegneria Informatica
Insegnamento di Software Architecture and Methodologies

Cosimo Giani - Davide Pucci

A.A. 2021 - 2022

Sommario

1 Introduzione

La banca **PSB** (**P**arri - **S**ampietro - **B**rizzi) ha commissionato una Home-Banking Application che permetta ad un Cliente di aprire e gestire un Conto Corrente comodamente da casa.
Nello specifico, il Cliente può scegliere tra diverse Tipologie di Conto Corrente offerte da PSB (conto Under30, conto Ordinario o conto Investitore) e aprire uno o più conti delle tipologie indicate. Ciascun conto si differenzia in termini di Costi, Tassi d'interesse e Massimali di prelievo.
Ad ogni conto può essere associata una o più Carte di diversa tipologia (Debito, Credito o Ricaricabile), dove ciascuna presenta un numero di carta, una data di scadenza e un massimale mensile.
All'interno dell'applicazione l'utente può visualizzare le Transazioni effettuate sul proprio conto e i loro dettagli.
Al momento della registrazione viene richiesto al Cliente di aggiungere i suoi dati anagrafici: questo può essere fatto caricando un PDF apposito da lui compilato che li contenga e che vengono estratti automaticamente dall'applicazione.
Per motivi di sicurezza PSB richiede che vi sia un'autenticazione a due fattori per accedere alla propria area personale.
Può essere attiva un'unica sessione alla volta: se un utente si logga da un altro device con un'altra sessione già aperta, quest'ultima si deve chiudere ed essere trasferita al nuovo dispositivo.
Qualora si desideri trasferire la sessione da un dispositivo all'altro, ad esempio da PC a mobile, sarà sufficiente inquadrare un QR apposito tramite la fotocamera.
A ciascun Cliente, al momento della registrazione, viene associato un Consulente bancario che si occupa di consigliare nuove offerte e fornire assistenza in caso di investimenti qualora il Cliente ne abbia necessità.
Un Consulente bancario può vedere i dettagli dei clienti cui è referente, così da poter fornire un supporto efficace.

1.1 Modulo di Testing

Al fine di valutare l'affidabilità e la sicurezza dei servizi online esposti dall'applicazione viene sviluppato un modulo di testing apposito. L'intento di tale modulo è quello di "stressare" i vari endpoint REST offerti, in modo

da garantire un livello di qualità soddisfacente e verificare l'efficacia del codice prodotto.
I risultati ottenuti da tale modulo verranno riportati a termine dello sviluppo.

2 Analisi dei requisiti software

In questa sezione vengono descritti i **requisiti software**, sono definite le **classi di utenza** e vengono modellati i **casi d'uso** fondamentali per l'applicazione.
Inoltre, sono riportati gli **obiettivi** che il committente vuole raggiungere.
I requisiti individuati sono stati derivati a partire dai vincoli e dagli obiettivi prefissati per questa applicazione, nel seguito riferita semplicemente con il termine **sistema**.

2.1 Macro Obiettivi - use case high summary

Gli obiettivi principali in merito alla richiesta di progettazione del nuovo sistema sono riassumibili nei seguenti punti:

- registrazione di un utente con conseguente apertura di un conto
- gestione del conto corrente con possibilità di visualizzazione dei dettagli, quali saldo e transazioni recenti
- gestione del profilo e account utente
- possibilità di fare affidamento su un Consulente Bancario

2.2 Classi di utenza

In questo paragrafo si descrivono le classi di utenza della futura applicazione, ossia le tipologie di utente che potranno accedere al sistema.

- **Correntista**, il quale rappresenta un utente che si è registrato con successo ed ha aperto un Conto Corrente.
- **Consulente**, il quale rappresenta la persona affiancata ad un Correntista che abbia necessità di assistenza.

2.3 Requisiti di sistema

I requisiti che seguono sono stati suddivisi in **3** categorie, a seconda della loro natura: **Requisiti di Dominio** (i.e., derivano dal dominio applicativo e sono espressi in termini di composizione di attributi - indicano le informazioni che si vuole rappresentare), **Requisiti Funzionali** (i.e., specificano cosa il sistema deve fare indipendentemente dalla tecnologie e dai mezzi implementativi utilizzabili - rappresentano le operazioni sulle informazioni) e **Requisiti Non Funzionali** (i.e., specificano cosa il sistema deve rispettare in termini di stabilità e affidabilità - non prescrivono cosa deve fare una capacità realizzata dal sistema, ma piuttosto come).

2.3.1. Requisiti di Dominio

- **RD-1**: una risorsa di tipo **Correntista** deve prevedere i seguenti attributi:
    - conti a lui associati
    - indirizzo e-mail
    - password
    - indirizzo di residenza
    - numero di telefono
    - anagrafica di base
    - identificativo Consulente
- **RD-2**: una risorsa di tipo **Conto** deve prevedere i seguenti attributi:
    - numero di conto
    - saldo
    - IBAN
    - insieme di transazioni effettuate
    - tipologia (Under30, Ordinario, Investitore)
    - carte associate
- **RD-3**: una risorsa di tipo **Dettagli Conto** deve prevedere i seguenti attributi:
    - costo, i.e. canone mensile
    - tasso di interesse
    - massimale di prelievo

- **RD-4:** una risorsa di tipo **Transazione** deve prevedere i seguenti attributi:
  - importo
  - data
  - luogo
  - tipologia (Bonifico, Versamento, Pagamento, Addebito)
- **RD-5**: una risorsa di tipo **Carta** deve prevedere i seguenti attributi:
  - tipologia (Credito, Debito, Ricaricabile)
  - numero di carta
  - data di scadenza
  - massimale
  - attiva/non attiva
- **RD-6**: una risorsa di tipo **Consulente** deve prevedere i seguenti attributi:
  - nome e cognome
  - indirizzo e-mail
  - matricola
  - password
  - clienti a lui associati
  - numero di telefono ufficio

2.3.2. Requisiti Funzionali

- **RF-1:** il sistema deve permettere di registrare un utente correntista
- **RF-2:** il sistema deve poter estrarre informazioni a partire da file PDF compilabili caricati, al fine di popolare istanze di dati
- **RF-3:** il sistema deve consentire di trasferire la sessione da un dispositivo all'altro, con un servizio dedicato (es. tramite QR)
- **RF-4:** il sistema deve consentire ad un Correntista di gestire i propri Conti Correnti:
  - **RF-4.1**: il sistema deve permettere la visualizzazione del saldo corrente
  - **RF-4.2**: il sistema deve permettere la consultazione delle transazioni effettuate e dei loro dettagli
  - **RF-4.3**: il sistema deve permettere la gestione delle carte associate

    - **RF-4.4**: il sistema deve permettere la visualizzazione delle tariffe previste dal proprio conto
    - **RF-4.5**: il sistema deve permettere la chiusura di un Conto Corrente
- **RF-5**: il sistema deve permettere ad un Correntista di gestire le proprie Carte:
    - **RF-5.1**: il sistema deve permettere la visualizzazione dei dettagli della Carta (numero, tipologia, data di scadenza, massimale)
    - **RF-5.2**: il sistema deve permettere di bloccare una Carta
- **RF-6:** il sistema deve permettere ad un utente non autenticato di compiere la procedura di autenticazione (i.e., *login*)
- **RF-7:** il sistema deve permettere ad un utente autenticato di effettuare la disconnessione dall'applicazione (i.e., *logout*)
- **RF-8**: il sistema deve consentire ad un Correntista di richiedere una consulenza
- **RF-9**: il sistema deve permettere ad un Consulente di visualizzare i dettagli dei clienti

2.3.3. Requisiti Non Funzionali

- **RNF-1:** il sistema deve essere implementato come un'applicazione web, fruibile tramite la rete Internet
- **RNF-2:** il sistema deve servirsi di un livello di persistenza che adotti un Database Management System di tipo relazionale
- **RNF-3:** il sistema deve essere realizzato su un'architettura RESTful
- **RNF-4:** il sistema deve implementare meccanismo di autenticazione
- **RNF-5**: il sistema di trasferimento della sessione utilizza un servizio REST e consente di trasferire da PC a mobile e viceversa.
- **RNF-6**: il sistema deve registrare un utente Correntista per mezzo di una operazione a tre passi:
    - **RNF-6.1**: inserimento credenziali di accesso, i.e. e-mail e password
    - **RNF-6.2**: inserimento dati utente di base
    - **RNF-6.3**: conferma apertura conto (via mail)

2.3.4. Vincoli Tecnologici

In accordo con il committente sono stati identificati i seguenti vincoli:

- il modulo di backend deve basare la sua implementazione sulle Specifiche Jakarta EE
- il sistema deve adottare come DBMS relazione MariaDb
- il sistema deve permettere il caricamento (upload) di file nel solo formato PDF compilabile

2.4 Casi d'uso

I casi d'uso vengono tipicamente adottati, in un processo di ingegneria del software, per documentare in modo compatto come la futura applicazione dovrà coprire i requisiti funzionali previsti per essa. L'analisi dei casi d'uso è importante per comprendere se, per ogni utente, attore di un caso d'uso, il sistema risponde alle sue esigenze funzionali; ossia la futura implementazione non avrà carenze in termini di funzionalità offerte che rispettano le richieste iniziali del committente.

2.4.1. Correntista

- **CU1** - Accedi "Login"
- **CU2** - Esci "Logout"
- **CU3** - Registrazione
    - **CU3.1** - Inserimento credenziali
    - **CU3.2** - Inserimento dati utente di base
        - **CU3.2.1** - Upload PDF compilabile
- **CU3.3** - Feedback apertura conto
- **CU4** - Seleziona Conto Corrente
- **CU5** - Gestisci Conto Corrente
    - **CU5.1** - Modifica dati personali
    - **CU5.2** - Visualizzazione transazioni
        - **CU5.2.1** - Visualizzazione dettagli transazione
    - **CU5.3** - Chiusura conto
    - **CU5.4** - Gestione carte associate
- **CU6** - Gestione Carta

    - **CU6.1** - Visualizzazione dettagli carta
    - **CU6.2** - Blocco carta
- **CU7** - Gestione sessione
    - **CU7.1** - Trasferimento sessione da PC a mobile
    - **CU7.2** - Trasferimento sessione da mobile a PC
- **CU8** - Visualizzazione dati Consulente
    - **CU8.1** - Richiesta consulenza

2.4.2. Consulente

- **CU1** - Accedi "Login"
- **CU2** - Esci "Logout"
- **CU9** - Visualizzazione lista Correntisti a lui associati
- **CU10** - Gestione Correntista
    - **CU10.1** - Visualizzazione conti
    - **CU10.2** - Visualizzazione anagrafica
- **CU11** - Gestione Conto Corrente di un Correntista
    - **CU11.1** - Modifica tipologia
    - **CU11.2** - Visualizzazione transazioni
    - **CU11.3** - Visualizzazione saldo
    - **CU11.4** - Gestione carte:
        - **CU11.4.1** - Aggiungi Carta
        - **CU11.4.2** - Rimuovi Carta
- **CU12** - Gestione Carta di un Correntista
    - **CU12.1** - Modifica massimali Carta

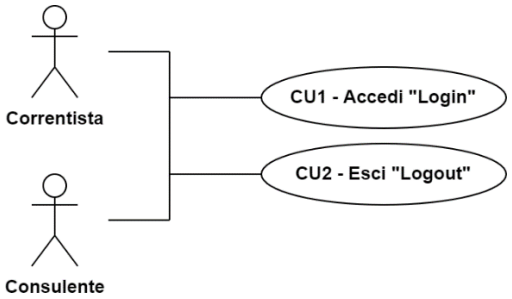

Figura 1 - Diagramma UML dei casi d'uso, gruppo "Autenticazione"

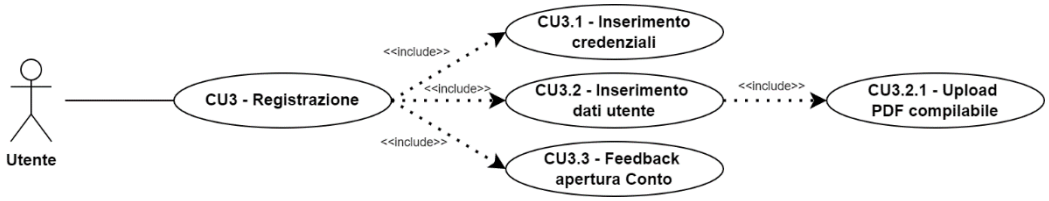

Figura 2 - Diagramma UML dei casi d'uso, gruppo "Registrazione"

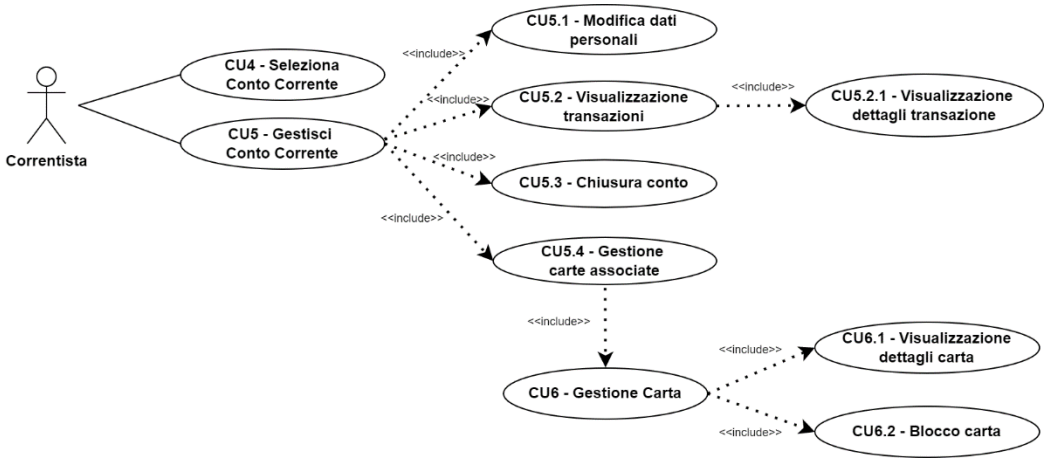

Figura 3 - Diagramma UML dei casi d'uso, gruppi "Conto Corrente" e "Carta"

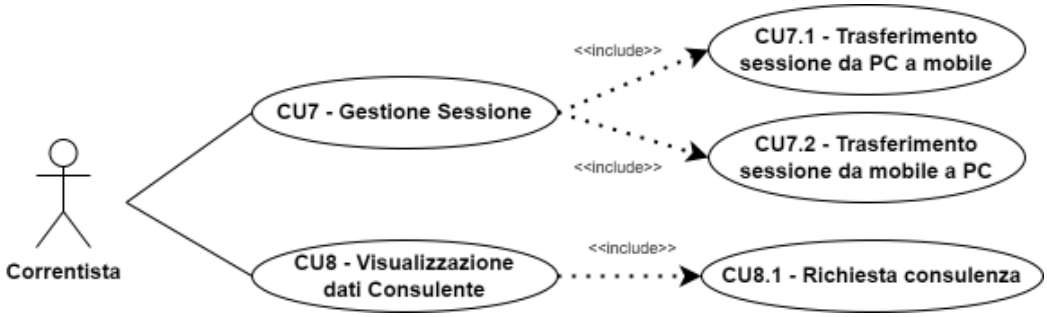

Figura 4 - Diagramma UML dei casi d'uso, gruppi "Sessione" e "Consulenza"

Figura 5 - Diagramma UML dei casi d'uso, gruppo "Gestione Correntisti"

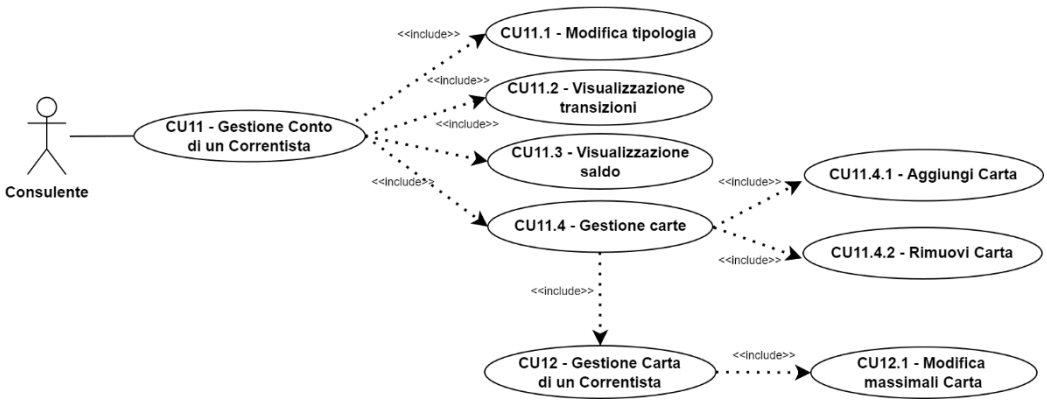

Figura 6 - Diagramma UML dei casi d'uso, gruppi "Conto" e "Carta" della risorsa "Correntista"

3 Modello di Dominio

Nello sviluppo software, un Modello di Dominio è un modello concettuale del dominio che incorpora sia il comportamento che i dati. Viene riportato in **Figura 7** il Domain Model dell'applicativo con tutte le entità coinvolte e le rispettive relazioni.

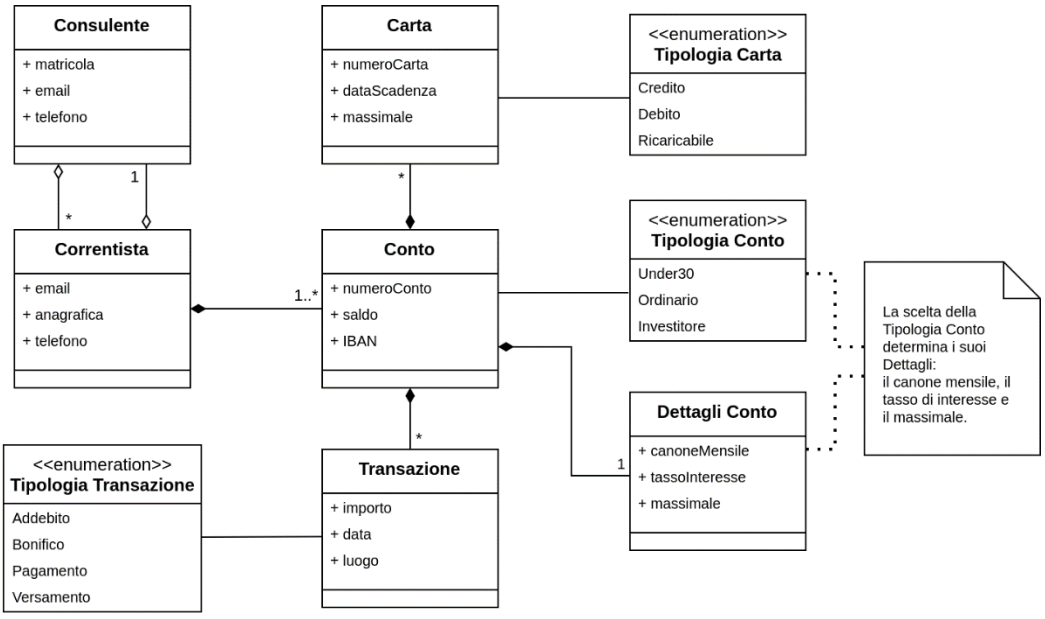

Figura 7 - Domain Model della Home Banking WebApplication. Al fine di semplificarelo schema non è stata inclusa la classe Base Entity per la gestione dello UUID.

Infine, viene riportato il deployment diagram dell'applicativo:

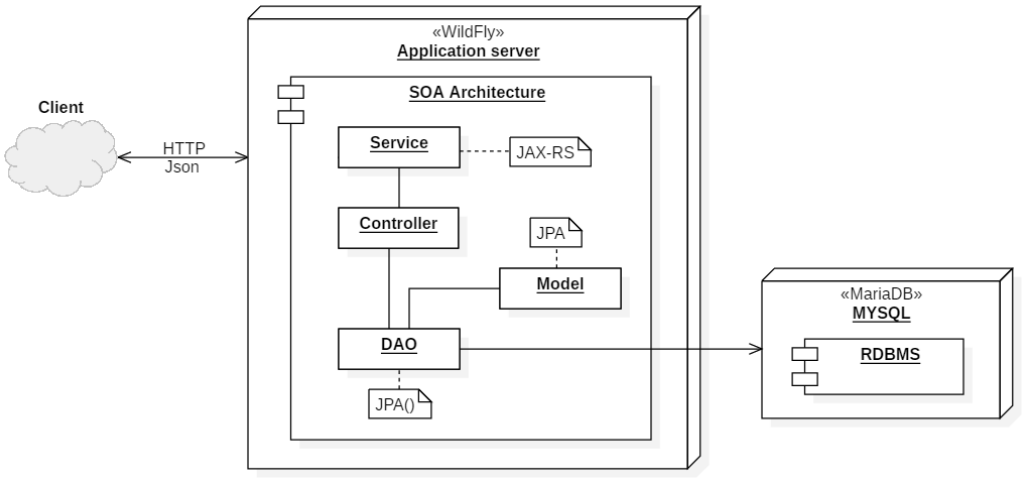

Figura 8 - Diagramma di deployment della WebApplication.

4 Moduli implementati

Sono riportate in questo capitolo le scelte riguardanti l'implementazione dei moduli principali del sistema con le opportune motivazioni.

4.1 Autenticazione Correntista e Trasferimento Sessione

L'autenticazione del Correntista avviene tramite verifica a due fattori. Viene esposto un servizio rest che, date e-mail e password del correntista, genera un codice OTP randomico che viene mandato via e-mail all'utente.
Quest'ultimo, dunque, in ogni richiesta HTTP che vorrà effettuare, offrirà nel campo *Authorization* dell'Header la sua e-mail e l'OTP che ha ricevuto. Lato backend sono state testate due policy per la generazione dell'OTP. Una prima *stateless* e una seconda *stateful*. Vengono esposti di seguito pro e contro di ciascuna opzione, anche in relazione al modulo di Trasferimento Sessione.

4.1.1. Autenticazione Stateless

La prima soluzione per la generazione dell'OTP è completamente stateless. Presa la password (criptata) dell'utente, questa viene concatenata con l'ora

corrente (ad esempio se sono le 17:30:22, si concatena il 17) e criptata tramite algoritmo MD5.

L'OTP così generato viene successivamente mandato via mail all'utente, che inserisce in un apposito campo nel frontend, così che venga utilizzato per ogni richiesta futura. Lato backend, quando arriva la richiesta, l'OTP viene generato nuovamente dalla password dell'utente e dall'ora corrente. Se l'ora non è cambiata (e se l'OTP nell'header è quello corretto) allora l'autenticazione avviene con successo.

In questo modo gli OTP non vengono salvati nel backend. D'altra parte, un approccio di questo tipo porterebbe ad una violazione del vincolo di **univocità** che la sessione possiede: per come viene generato l'OTP, questo ha una durata temporale di un'ora, di conseguenza un meccanismo di autenticazione basato sull'orario permetterebbe a più device (autenticati) in contemporanea di mantenere la sessione attiva.

4.1.2. Autenticazione Stateful

Per quanto riguarda la strategia *stateful* viene utilizzato un Bean *ApplicationScoped* che mantiene delle coppie <e-mail - OTP>. Quando un Correntista richiede la generazione di una password temporanea per autenticarsi, invia ad un servizio rest dedicato e-mail e password e riceve indietro via mail l'OTP.
Nello specifico, questo OTP viene generato randomicamente e gli viene associato un timer (di 5 minuti), alla cui scadenza l'OTP viene eliminato. Ogni volta che il Correntista desidera accedere ad uno dei servizi rest dell'applicazione, come in precedenza, dovrà fornire il proprio OTP nell'header della richiesta. Ad ogni richiesta effettuata la validità temporale dell'OTP si aggiornerà, resettando così il suo tempo di vita a 5 minuti. Secondo questa logica, dopo 5 minuti di inattività l'utente viene disconnesso dall'applicazione.

Questo approccio permette di generare un OTP completamente nuovo qualora si desideri trasferire la sessione su un dispositivo differente da quello in uso, rispettando così il vincolo di **univocità**. Inoltre, il fatto che l'OTP abbia una durata di massimo 5 minuti e che sia totalmente randomico permette

di rendere più sicuri i servizi esposti. Di contro, adottando questa modalità, si va incontro ad una violazione della *statelessness* che caratterizza l'architettura RESTful su cui l'applicativo oggetto di questo elaborato si basa.

A seguito di un'attenta riflessione su quanto appena riportato, congiuntamente alla necessità di gestire al meglio il meccanismo di trasferimento della sessione, è stato deciso di adottare un'**architettura RESTful ibridata**, in cui le richieste vengono svolte in modalità *stateless* tra client e server, ma in cui la verifica di queste è basata su una componente *stateful*.

L'autenticazione per il Consulente è gestita analogamente. L'unica differenza è che non viene utilizzato un OTP mandato via mail all'Utente, ma si fa uso di un Token ritornato al Client quando la login viene eseguita correttamente.
Tale Token deve essere presentato ad ogni successiva richiesta, assieme alla matricola del Consulente. Come l'OTP il Token si invalida dopo 5 minuti di inattività.
Non è prevista una logica di trasferimento di sessione per il Consulente.

Nell'appendice A in **Figura 10** sono riportate in dettaglio le interazioni tra oggetti al momento dell'autenticazione, rappresentate attraverso il formalismo degli UML Sequence Diagram.

4.1.3. Trasferimento di Sessione

Il modulo di Trasferimento della Sessione espone due servizi rest:

- `session/request-transfer`
  Date e-mail e OTP valide, fornisce un codice ad 8 cifre che deve essere utilizzato nel servizio **get-transfer-credentials** entro un minuto per trasferire la sessione.
- `session/get-transfer-credentials`
  Dato il codice ad 8 cifre generato al passo precedente, ritorna e-mail e OTP valide per l'autenticazione dell'utente che ha richiesto il trasferimento della sessione.

Questi due servizi si appoggiano su un Bean *ApplicationScoped* denominato **ServiceTransferManager**. Questa classe mantiene in una mappa privata delle coppie <*codiceSessione* - *credenzialiUtente*> che vengono create quando il servizio `request-transfer` viene chiamato. Tali coppie vengono eliminate dopo un minuto dalla creazione tramite dei TimerTask dedicati. Entro un minuto, perciò, facendo uso del servizio apposito, è possibile recuperare le credenziali utente inserendo il *codiceSessione*. Una volta inserito il *codiceSessione* la coppia viene cancellata, così che non possa essere utilizzato il codice di trasferimento più di una volta.

Utilizzando la strategia descritta per la generazione dell'OTP, quando si procede con il trasferimento della sessione, un OTP completamente nuovo viene generato e associato. Così facendo la vecchia sessione non risulta più valida e viene mantenuto il vincolo di **univocità** della sessione.

Nell'appendice A in **Figura 11** sono riportate in dettaglio le interazioni tra oggetti al momento del trasferimento della sessione, rappresentate attraverso il formalismo degli UML Sequence Diagram.

4.2 Elaborazione dati PDF

Come espresso dal requisito funzionale **RF-2** è stato implementato un modulo per l'estrazione delle informazioni a partire da file PDF compilabili.
Al momento della registrazione è richiesto il caricamento di un PDF opportunamente compilato, che viene inviato al back-end tramite il servizio rest `registration/send`.
Il controller di tale servizio si interfaccia con `PdfUtil`, il modulo che si occupa di elaborare il PDF caricato. In dettaglio, quest'ultimo salva sul server il PDF caricato, dunque fa uso del package `e-iceblue.spire.pdf.free` per estrarre i dati. Infine, elimina dalla memoria il PDF caricato, dato che i dati saranno già stati estratti e salvati sul DB. Grazie alle funzionalità del package utilizzato, siamo in grado di individuare i campi compilabili ed estrarne il valore.

È presente un'attenta logica di controllo che si occupa di verificare che il PDF caricato sia effettivamente il modulo di registrazione e che tutti i campi

siano compilati correttamente. Nella fase di testing tutti questi scenari sono stati opportunamente validati.

Nell'appendice A in **Figura 9** sono riportate in dettaglio le interazioni tra oggetti al momento della registrazione, rappresentate attraverso il formalismo degli UML Sequence Diagram.

4.3 Testing

Al termine dell'implementazione dell'applicativo di home banking ha avuto luogo una fase di testing.
L'obiettivo di questa procedura è stato, da un lato, di verificare che ciascun elemento architetturale dell'applicazione funzionasse correttamente, dall'altro, di andare a "stressare" i vari endpoint REST offerti, cercando di porre l'attenzione su quegli aspetti di sicurezza e di comportamento propri dell'applicazione.

In questo contesto, per poter efficacemente testare il codice prodotto, è stato fatto uso dei seguenti framework:

- **JUnit 5**, come strumento per Unit testing in Java
- **Mockito**, a supporto del testing che mette a disposizione un meccanismo alternativo per la gestione di aspetti quali l'injection
- **RestAssured**, al fine di testare gli Endpoint e verificarne il comportamento complessivo è necessario fare uso di una libreria del genere per la definizione e l'uso degli URL

La realizzazione di un'architettura RESTful ibridata, come esposto nel paragrafo 4.1.2, ha portato inoltre a dover tenere in considerazione l'occupazione di memoria per mantenere le informazioni dei vari utenti loggati. A questo scopo la fase di test è stata complementata con uno studio delle performance dell'applicativo al variare del numero di utenti loggati. Nel paragrafo 4.3.3 sono riportati ed analizzati i risultati ottenuti.

4.3.1. Unit test

Con l'obiettivo di voler validare il codice prodotto, sono stati implementati una serie di test atti a verificare la correttezza dei vari componenti architetturali.
Nello specifico sono stati utilizzati dei Test di Unità al fine di verificare l'integrità comportamentale del *domain model*, dei *DAO* e dei *controller*. Inoltre, è stato verificato per il contesto operativo d'esercizio il *modulo di import dati* (pre-compilati) da documenti in formato PDF.

- **Domain model**
  Nella procedura di testing, per quanto riguarda il model, è opportuno testare solo gli elementi non banali. Dal momento che l'applicazione presenta una superclasse adeguata chiamata *BaseEntity* è stato deciso di testare unicamente tale classe andando a verificare la corretta inizializzazione dei vari oggetti e per evitare di dover fare minimi test su ogni Entità. La difficoltà in questo caso risiede però nel fatto che *BaseEntity* è una classe astratta: per questo motivo è stato deciso di implementare un'estensione identica ma concreta di suddetta classe da inserire poi nel package di test, ovvero *FakeBaseEntity*.

- **Data Access Object (DAO)**
  Per poter testare i DAO è necessario predisporre un apposito Database dedicato, i.e. un *in-memory database*. Dato che però per poter testare efficacemente questi componenti è necessario fare alcune inizializzazioni non banali è stato deciso di implementare convenientemente una classe astratta *JPATest* che sarà poi estesa da ogni classe di test per i DAO e che permette di sfruttare i vari hook predisposti da JUnit.

- **Controller**
  Questi componenti fanno frequente uso di dipendenze esterne (i.e. DAO e entità). Per questo motivo è fondamentale l'uso dei *mocks*. Quando i controller ricevono per injection un componente, al fine di testare il controller in isolamento, è necessario fare uso dei componenti di mock - con

cui si definisce un'istanza di mock e il relativo comportamento manualmente - e dell'injection manuale - con cui si inietta manualmente l'istanza appena creata nel controller.

- **Modulo di import dati da PDF**
  Questo modulo, oltre a sfruttare i mock per le dipendenze esterne, fa uso di una serie di file .pdf per fare la validazione dell'estrazione e dell'import dei dati contenuti in suddetti file. Questi documenti sono stati realizzati ad hoc per verificare e coprire tutti gli scenari di errore che si possono presentare a seguito di una errata compilazione del PDF. Infine, viene verificato che la mappa estratta per mezzo di questo modulo sia coerente con i dati del documento.

4.3.2. Integration test

Al fine di testare gli endpoint REST in uno scenario realistico ed attento alla sicurezza sono stati effettuati dei test di integrazione.
Per fare ciò viene deployata l'applicazione in un server locale WildFly, utilizzando come datasource uno schema di test su MariaDB. In questo modo è possibile chiamare i servizi rest dell'applicazione deployata tramite JUnit ed essere sicuri che tutti i componenti si comportino come da aspettativa.
JUnit può caricare delle istanze di test direttamente sul DB facendo uso di JDBC: così facendo è possibile testare tutti gli scenari di utilizzo dei servizi rest, assicurando che questi siano sempre opportunamente protetti.
Per facilitare la configurazione della suite di test viene fatto uso di una classe `ServiceTest` che si occupa di:

- gestire la configurazione di RestAssured ed eseguire le richieste agli endpoint una volta formulate
- gestire la connessione al DB di Test, così che possano essere caricate le istanze di test
- formalizzare la struttura dei test, offrendo un'interfaccia comune a tutti i test specifici, mantenendo così un discreto grado di riusabilità

Alla classe è associato un file `configurations.json` in cui sono presenti i dati di configurazione relativi all'implementazione. Nello specifico sono presenti:

- "baseURI" e "port" da associare a RestAssured
- "connectionURL" per il DB di test e credenziali di accesso ("username" e "password")
- "deploymentPath" del server locale, in modo da poter accedere ai file salvati sul server qualora ve ne sia necessità durante i test

Tale file viene inserito nelle *resources* di test e caricato dalla classe `ServiceTest` prima che i test vengano eseguiti (grazie all'annotazione JUnit `@BeforeAll`). Nello specifico si verifica che i dati inseriti nel file siano corretti, andando a controllare che:

1. l'URI indicata sia raggiungibile con una *get()* sulla porta specificata
2. la connessione al DB avvenga con successo tramite JDBC

I test specifici vengono implementati estendendo la classe `ServiceTest` e facendo uso dell'annotazione JUnit `@Test`. Grazie al metodo astratto della classe base `beforeEachInit()` vengono offerte alle sottoclassi una modalità strutturata per l'inizializzazione, dove, ad esempio, è possibile fare uso della connessione al DB per caricare le istanze di test.
È stata inoltre implementata una classe `QueryUtils` per incapsulare le query di inizializzazione che sono state utilizzate nei vari test.
Infine, è presente una classe `OTPUtils` che si occupa di "intercettare" l'OTP di accesso degli utenti una volta generata, in modo da poter testare i servizi rest in uno scenario reale in cui l'OTP fornita è proprio quella generata dal backend e mandata via mail a chi richiede l'accesso.
Lo schema del package di test implementato è mostrato in **Figura 9** nell'appendice A, mentre in **Figura 10** è rappresentato lo schema strutturale dei test per le chiamate ai servizi rest.

Questo modulo di test, nel binomio costituito da `ServiceTest` e dal file `configurations.json`, presenta un alto grado di configurabilità e offre tutte le funzionalità necessarie per un test di integrazione che vada a stressare esaustivamente gli endpoint rest di una applicazione web. Sarebbe possibile testare una qualsiasi web application con questo modulo, sia che questa sia deployata in locale che questa sia in un server esterno, a patto che sia

possibile stabilire una connessione sul DB sottostante all'applicazione, in modo da poter caricare delle istanze di test.

Sono stati testati minuziosamente tutti i servizi rest esposti dall'applicazione, verificando che l'OTP venga generato esclusivamente quando le credenziali sono corrette, che questo risulti essere valido solamente per l'utente che ha richiesto l'accesso, che il trasferimento di sessione sia sicuro, che sia possibile accedere esclusivamente a informazioni e oggetti di proprietà dell'utente autenticato e che non vi siano problematiche di SQL Injection al momento dell'autenticazione.

4.3.3. Performance test

Come anticipato, la realizzazione di un'architettura RESTful ibridata ha portato a dover tenere in considerazione l'occupazione di memoria per il mantenimento di alcune informazioni relative ai vari utenti loggati, nello specifico i dettagli sulla sessione.
Un approccio di questo tipo, pertanto, implica di dover "tenere in vita" molti componenti, come può accadere in uno scenario aziendale moderno.

Per il motivo appena citato, l'ultima analisi effettuata riguarda l'impatto in termini di risorse dei moduli di autenticazione e di trasferimento sessione, in quanto fanno uso di una componente di memorizzazione.
Nello specifico viene effettuato uno studio dell'occupazione di memoria di questi servizi al variare del numero di utenti che ne fanno uso.

Nel caso del servizio di autenticazione viene valutato il footprint del bean *ApplicationScoped* che memorizza le coppie <e-mail - OTP> degli utenti autenticati. In **Figura 14** si può notare come l'occupazione di memoria scali linearmente con il numero di utenti.

Come massimo carico in questa fase vengono testati 5 milioni di utenti autenticati in contemporanea, con un'occupazione di memoria di circa **2.18** GB.
È possibile concludere che il servizio di autenticazione in questa modalità riesce a sopportare un carico di lavoro di oltre 5 milioni di utente con 4GB di RAM a disposizione per la JVM.

È stato effettuato un test analogo per il servizio di trasferimento della sessione, in quanto anche in questo caso viene fatto uso di un bean *ApplicationScoped* che memorizza coppie <codiceSessione - credenzialiUtente>.
In **Figura 15** sono riportati i risultati ottenuti al variare del numero di richieste di trasferimento effettuate. In questa fase sono state testate fino a 10 milioni di richieste in contemporanea, con un'occupazione di memoria pari a circa **3.7** GB.

Entrambi i test di massimo carico possono risultare eccessivi per lo scenario di utilizzo dell'applicazione. 10 milioni di richieste di trasferimento sessione in contemporanea non rappresenta un numero verosimile, mentre il numero di utenti autenticati potrebbe anche avvicinarsi ai 5 milioni nel contesto di aziende multinazionali.

In **Figura 16** viene riportato un confronto tra i due servizi. Si può notare come il servizio di autenticazione risulti lievemente essere più impattante rispetto a quello di trasferimento. Questo è dovuto al fatto che nel primo servizio viene memorizzato anche il tempo di validità dell'OTP, mentre nel secondo si ha sempre 1 minuto a disposizione per trasferire la sessione. Entrambi i servizi scalano linearmente con il numero di utenze; perciò, è possibile effettuare una stima accurata delle risorse necessarie al corretto funzionamento dell'applicazione, compatibilmente con il carico atteso.

Qualora non fosse possibile offrire le risorse necessarie, si potrebbe fare uso di un servizio esterno che permetta di avere una coda delle richieste, in modo da evitare sovraccarichi e mitigare il problema presentato in questa sezione.

Infine, per avere conferma che i servizi liberassero le risorse correttamente una volta utilizzate, è stato effettuato un test di login/logout su 10.000 utenti. In **Figura 17** si può osservare come con l'aumentare del numero di utenti loggati l'utilizzo di memoria cresca linearmente, mentre, quando questi si disconnettono, si ha un decremento lineare dell'utilizzo di RAM, fino a tornare a valori usuali.

Appendice A

- Class Diagram

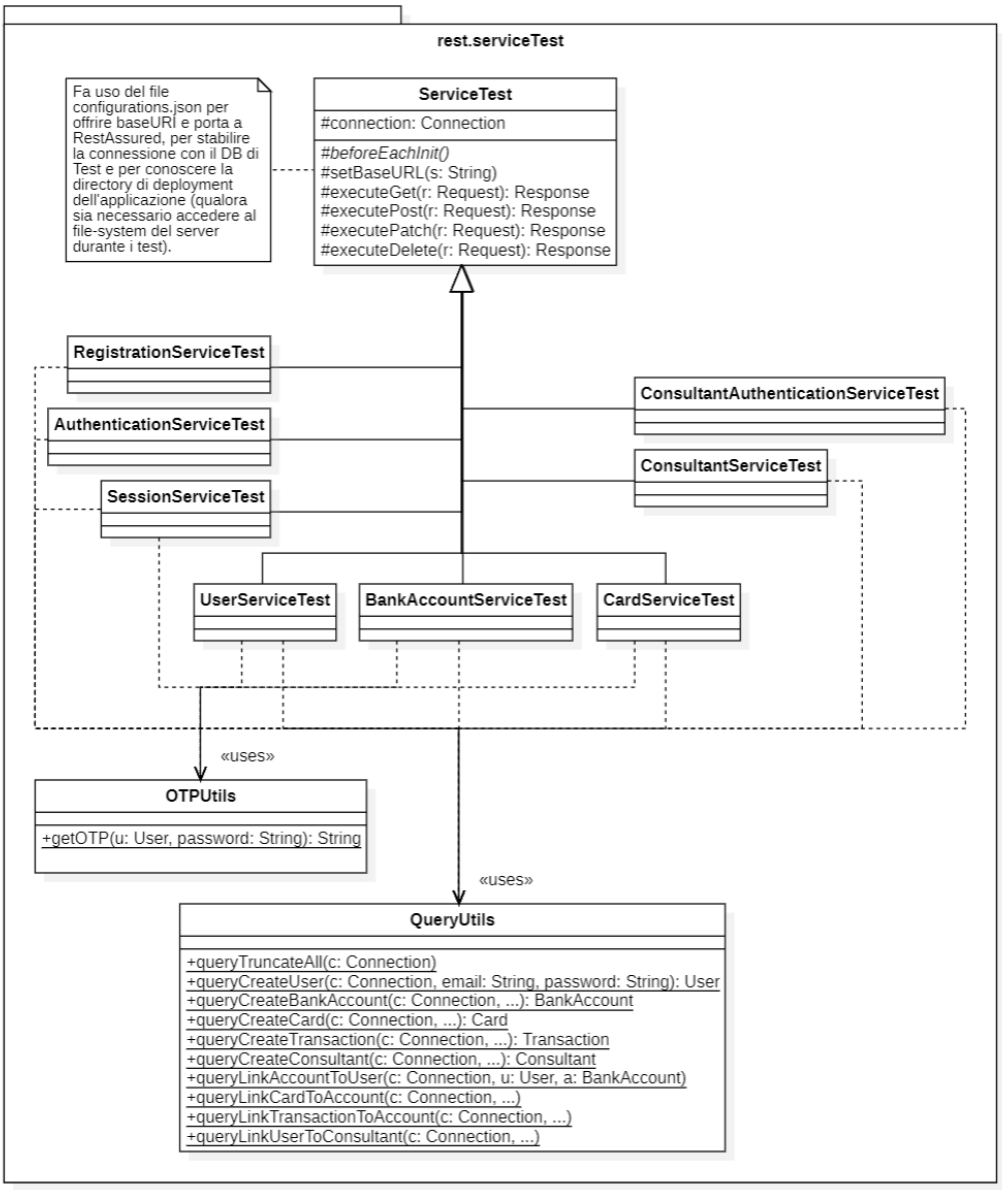

Figura 9 - Diagramma di classe per ServiceTest.

- Schema strutturale dei test

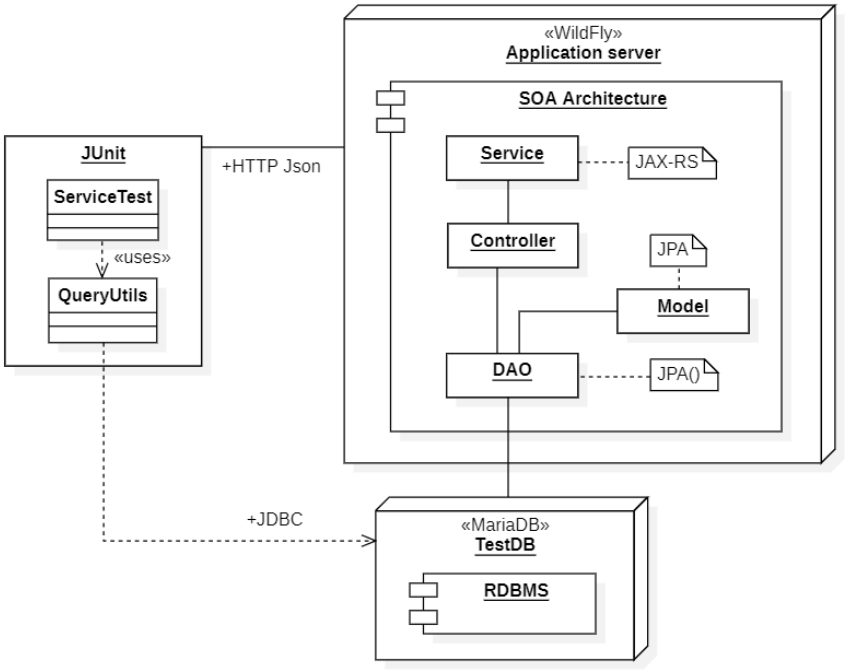

Figura 10 - Schema strutturale dei test nel caso di Integration Test per i servizi "rest".

- Sequence Diagram

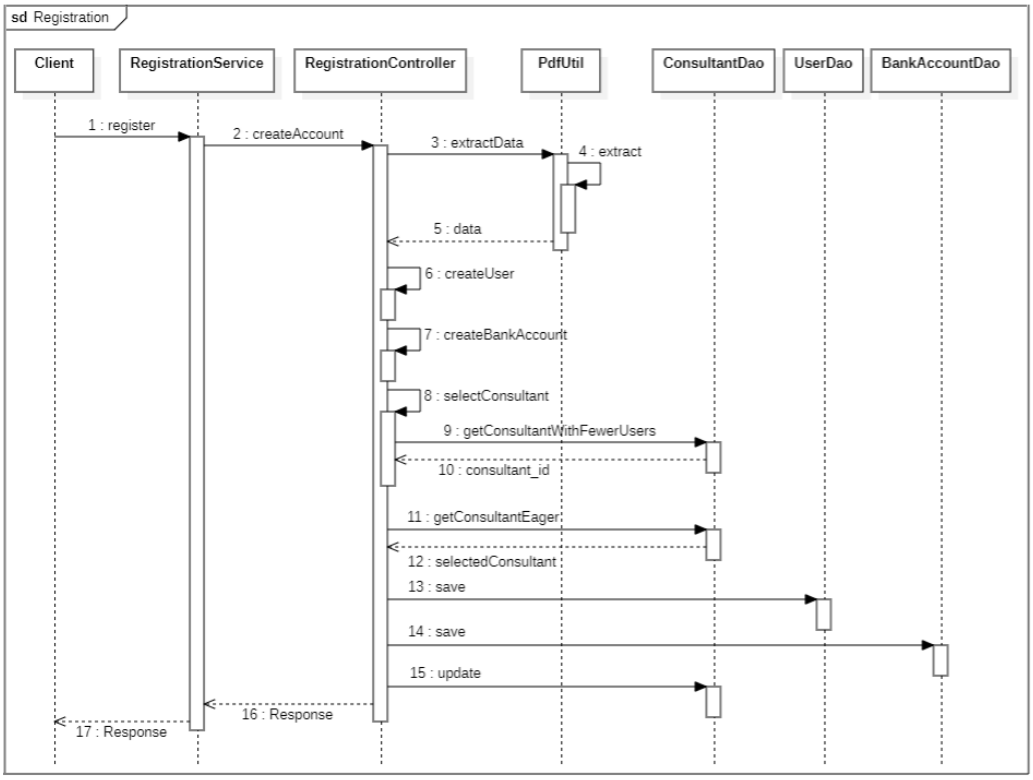

Figura 11 - Diagramma di sequenza per il servizio di registrazione.

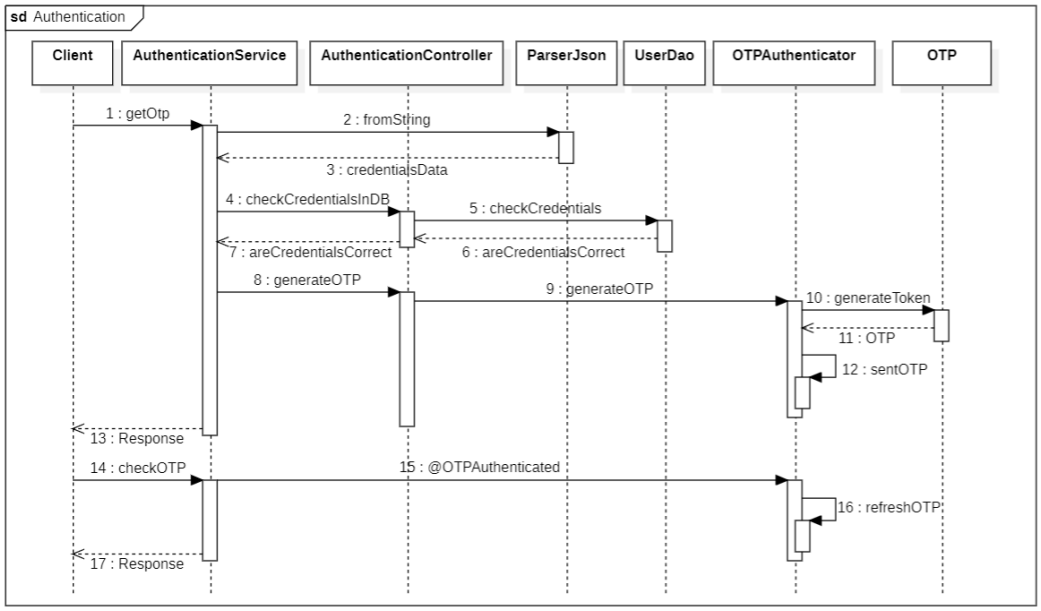

Figura 12 - Diagramma di sequenza per il servizio di autenticazione.

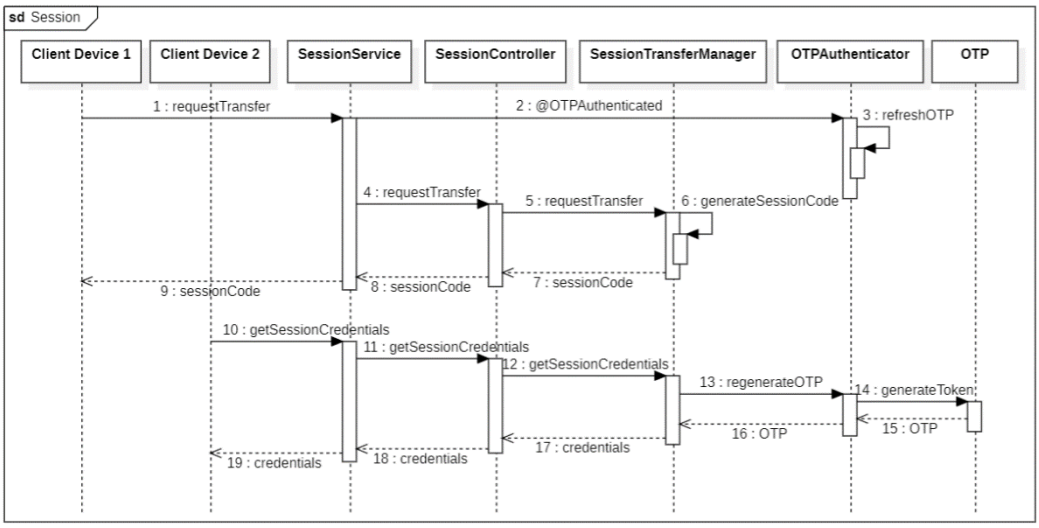

Figura 13 - Diagramma di sequenza per il servizio di trasferimento della sessione.

- Grafici

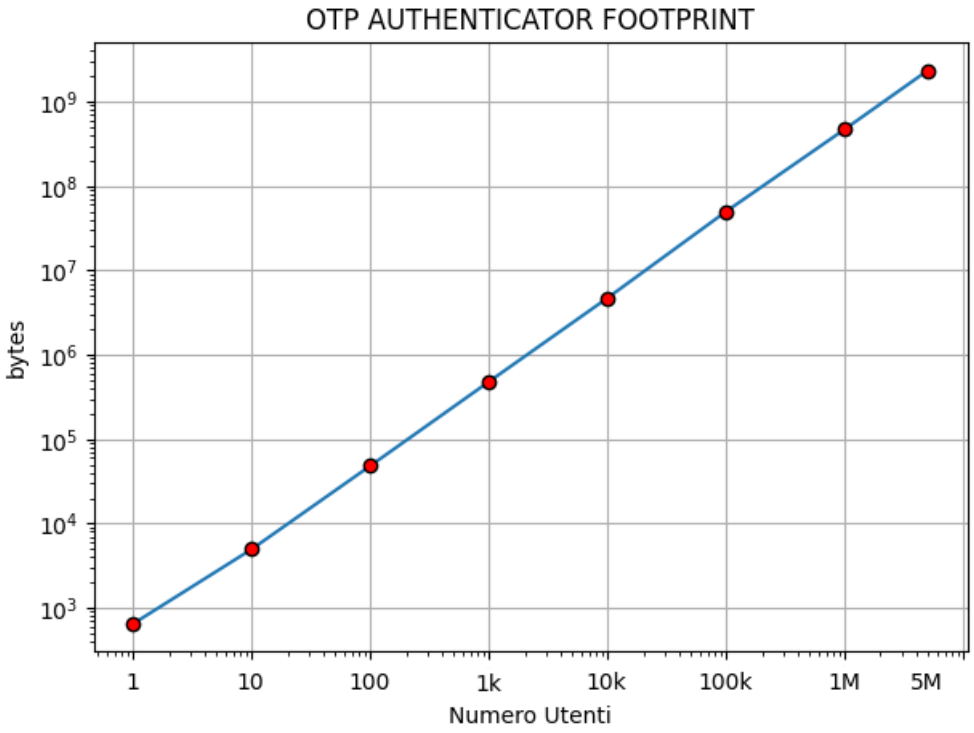

Figura 14 - Grafico dell'utilizzo di memoria al variare del numero di utenti per il servizio di autenticazione tramite OTP.

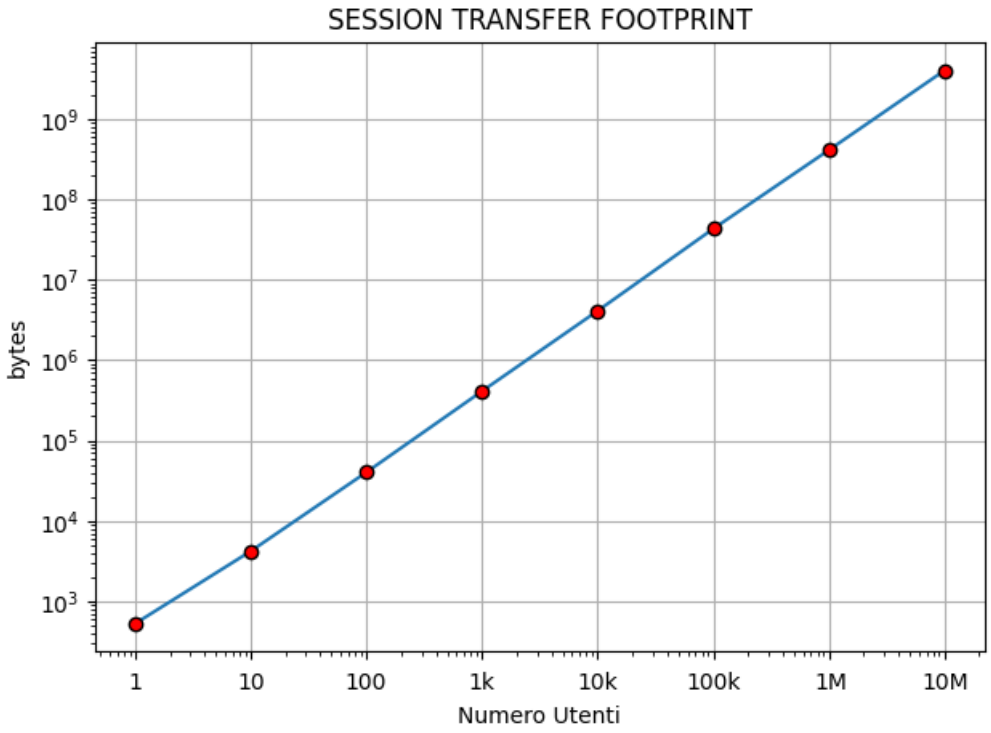

Figura 15 - Grafico dell'utilizzo di memoria al variare del numero di utenti per il servizio di trasferimento della sessione.

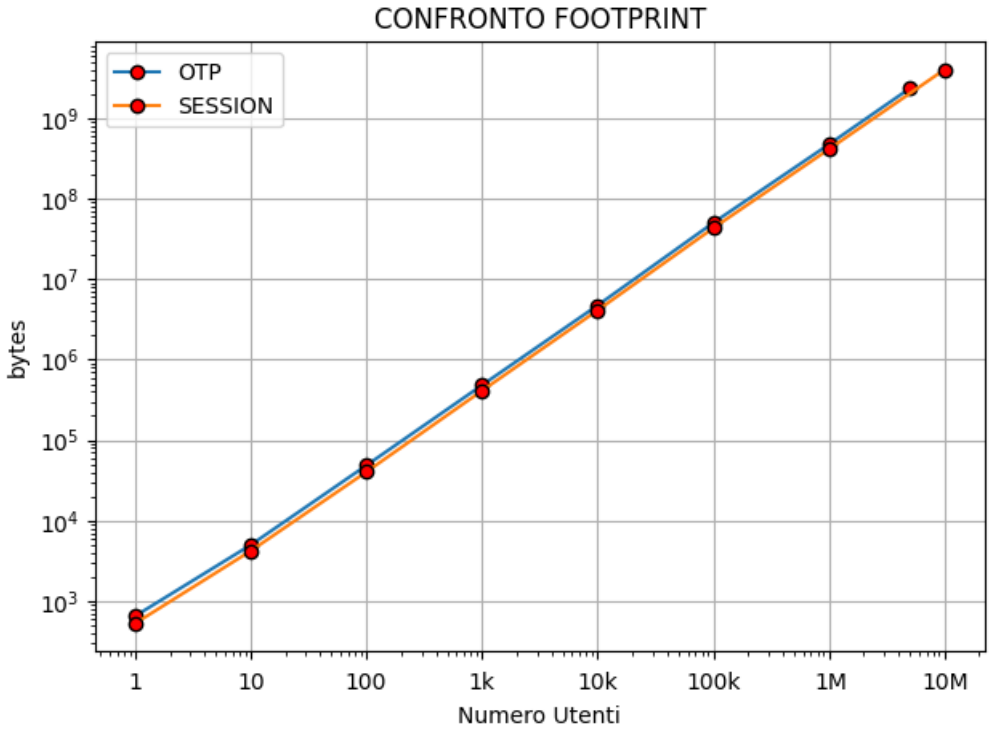

Figura 16 - Confronto servizi OTP e Session in termini di utilizzo di memoria.

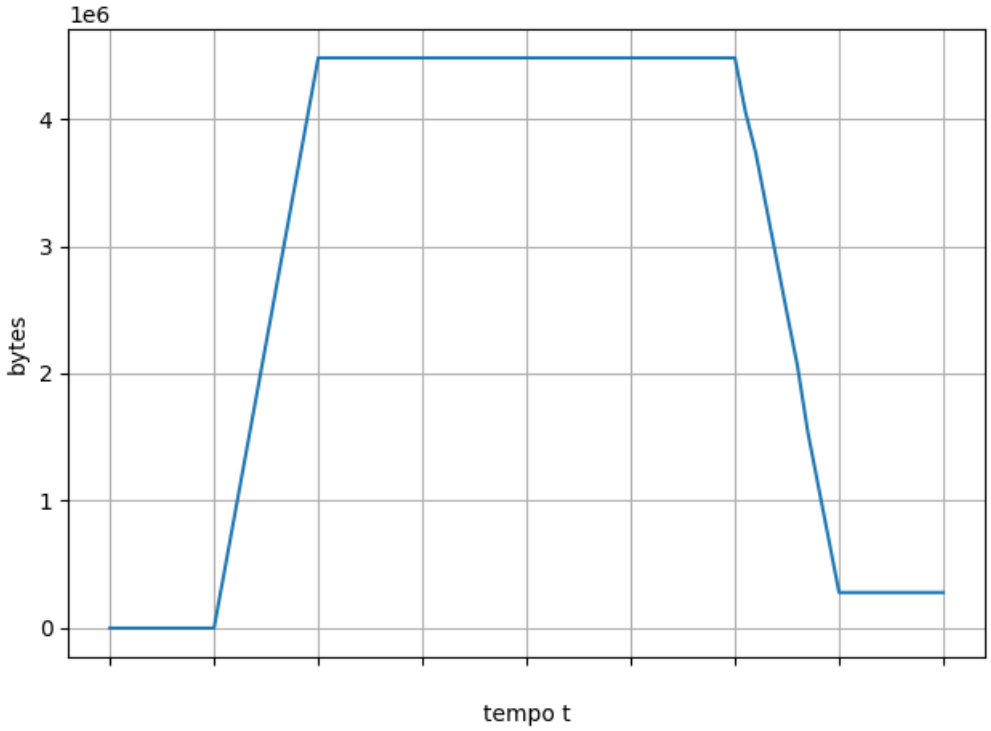

Figura 17 - Grafico dell'utilizzo di memoria nel tempo nel caso di login e successivo logout di 10.000 utenti.