Politecnico di Torino

Corso di Laurea Magistrale in
INGEGNERIA INFORMATICA

Sviluppo e Integrazione di Microservizi nel settore Bancassurance con Pattern Sincroni/Asincroni e GenAI

Politecnico
di Torino

Candidato:
Giovanni Battista CESTA

Relatore:
Dr. Riccardo COPPOLA

Anno Accademico 2024/2025

Sommario

Nell'ultimo decennio la diffusione di device personali come smartphone, tablet e PC ha portato il settore bancario e assicurativo a spingersi verso servizi digitali basati su architetture software moderne, scalabili e che offrano sicurezza. Negli ultimi anni, sempre più persone hanno familiarizzato con i servizi digitali bancari, imparando a gestire le proprie finanze direttamente da smartphone o PC. Complice l'esperienza del lockdown, che ha reso impossibile recarsi fisicamente in filiale, molti hanno scoperto la comodità di operare online, riducendo così la necessità di lunghe attese agli sportelli. Questa tesi si pone l'obiettivo di sviluppare un'applicazione basata su microservizi per il settore bancario assicurativo per offrire ad ogni utente diversi canali online su cui sarà possibile operare in completa autonomia. Per raggiungere questi obiettivi è stato valutato di sfruttare tecnologie avanzate come Spring Boot, REST API, connettori Kafka e agenti AI. È stato scelto di sviluppare questo progetto seguendo un approccio Waterfall, basandosi su una documentazione dettagliata di tutte le fasi di sviluppo che ne permetta una gestione controllata. Le scelte architetturali si concentrano sulla progettazione di un sistema formato da più moduli, resiliente e scalabile, che rispetti le esigenze di sviluppo in ambito finanziario, come la gestione delle transazioni, della sicurezza e dell'integrità dei dati. Sono stati analizzati i pattern architetturali più idonei al sistema in esame per sfruttare le potenzialità di ognuno di essi: la comunicazione asincrona basata su eventi con Kafka, l'esposizione di microservizi tramite paradigma REST, l'uso di State Machine e l'ottimizzazione del ciclo di vita dei dati. Sono stati presi in considerazione due differenti approcci nei sistemi distribuiti come orchestrazione e coreografie funzionali, evidenziandone i punti di forza e l'utilizzo in processi complessi come la generazione di documenti o la gestione della firma digitale. Inoltre, si è scelto di utilizzare alternativamente la comunicazione sincrona (API REST) e asincrona (Kafka), evidenziando i vantaggi in termini di affidabilità e prestazioni. Nella fase di test sono stati utilizzati Copilot, per velocizzare la fase di scrittura di test ripetitivi, e Agent AI, nella fase di troubleshooting, per facilitare la ricerca di issue in produzione. L'implementazione dell'intero sistema e l'uso di tecnologie all'avanguardia offrono una nuova panoramica sullo sviluppo di software in ambito finanziario per permettere l'utilizzo di architetture a microservizi che svolgono un ruolo fondamentale nel settore assicurativo, il quale richiede elevati livelli di sicurezza, prestazioni e disponibilità.

Indice

Elenco delle figure

Listings

Capitolo 1

Introduzione

Questa tesi è stata scritta con l'obiettivo di illustrare il funzionamento generale del progetto che abbiamo realizzato, evitando qualsiasi riferimento esplicito all'azienda che lo ha commissionato. Perciò tutti i nomi associati al prodotto "polizza a protezione del credito", gli stati della State Machine e le definizioni e i corpi delle funzioni, sono stati inventati o opportunamente modificati per garantire l'anonimizzazione delle informazioni aziendali.
Il flusso di vendita e le coreografie funzionali sono state semplificate al minimo indispensabile, mantenendo soltanto gli elementi essenziali per la comprensione del processo. Questo approccio consente di illustrare in modo chiaro e conciso il flusso delle operazioni senza entrare nei dettagli specifici o nei servizi interni coinvolti nella realizzazione del prodotto finale. L'obiettivo è mostrare una panoramica generale del progetto rispettando la riservatezza dell'azienda e concentrandosi sulle scelte architetturali del sistema.
L'architettura a microservizi rappresenta una delle principali innovazioni nello sviluppo di applicazioni moderne, in particolare nel settore bancario e assicurativo, dove l'integrazione di diversi sistemi è cruciale per offrire soluzioni multi-canale. L'uso di tecnologie come Java, Spring e Spring Boot per la creazione di API REST consente di facilitare comunicazioni sincrone tra sistemi, garantendo al contempo scalabilità e modularità. Tuttavia, per soddisfare esigenze di comunicazione asincrona e real-time, risulta essenziale l'impiego di connettori Kafka, che permettono una gestione efficiente dei flussi di dati tra sistemi distribuiti. L'integrazione sinergica di queste tecnologie consente la vendita di prodotti assicurativi attraverso diversi canali, fornendo un'esperienza utente ottimizzata e flessibile.
Questa tesi si pone l'obiettivo di sviluppare ed evolvere microservizi che integrino sistemi bancari e assicurativi, utilizzando API REST per le comunicazioni sincrone e Kafka per quelle asincrone. Per facilitare la programmazione, verranno utilizzati strumenti basati su intelligenza artificiale generativa, co-

me GitHub Copilot, che supporteranno lo sviluppo del codice in maniera efficiente e rapida. Inoltre, la gestione dei task e della collaborazione tra i membri del team verrà eseguita tramite piattaforme come Notion, favorendo una gestione agile e centralizzata delle informazioni e delle attività. Saranno realizzati diversi connettori che gestiranno i flussi di informazioni tra i sistemi, con particolare attenzione alla sicurezza, alla scalabilità e all'affidabilità dell'infrastruttura.
La tesi è divisa in cinque capitoli, ognuno dei quali tratta un diverso aspetto dello sviluppo del progetto. Nel primo capitolo c'è l'introduzione nella quale sono stati spiegati lo scopo e gli obiettivi del lavoro svolto. Nel secondo sarà descritto il background tecnologico che andremo ad utilizzare spiegando le scelte prese in base alle richieste presentateci dal cliente. Nel terzo capitolo verrà illustrato come queste scelte sono state implementate utilizzando tecnologie moderne e scalabili. Nel quarto saranno esposte sia le strategie di testing adottate per avere un prodotto efficiente e limitare le operazioni di backtracking nel caso di malfunzionamenti, sia il meccanismo di troubleshooting semplificato grazie all'uso di genAI. Infine nel quinto e ultimo capitolo verranno mostrate le conclusioni e i possibili sviluppi futuri su cui si potrà lavorare.

Capitolo 2

Background Tecnologico

2.1 Fasi di sviluppo

L'obiettivo della tesi è lo sviluppo di un'applicazione basata su microservizi che si occupi della vendita e della gestione del post-vendita di una 'Polizza a protezione del credito'. Ogni nuova polizza che un utente ha intenzione di acquistare è collegata ad un prestito già finanziato dalla banca stessa. Fino ad adesso, per ottenere questo prodotto era necessario presentarsi presso una filiale ed essere assistiti da un gestore che, usando un canale interno, finalizzava l'acquisto. Con questa nuova applicazione multicanale (web, mobile app, desktop app e filiale) sarà possibile completare in completa autonomia l'acquisto della polizza. Per il nostro prodotto è stato scelto di sviluppare, in accordo con il cliente, il ciclo di vita del progetto secondo la metodologia Waterfall.

2.1.1 Waterfall vs Agile

In base alle richieste si è pensato che questa modalità fosse la più adatta in questo contesto. La scelta è stata presa tenendo in considerazione la disponibilità economica e delle risorse. A differenza di Agile, che richiede una continua assistenza e revisione dei requisiti, il metodo Waterfall permette di accordarsi sin dall'inizio sui requisiti e sui dettagli di implementazione [2]. Questo metodo consente di produrre un software seguendo uno sviluppo lineare separato in fasi ben distinte. È molto importante avere una documentazione precisa e ben dettagliata di ogni passaggio del processo. In essa devono essere specificati tutti i requisiti che dovremo affrontare e i cambiamenti al piano iniziale dovranno essere minimi. Le fasi si possono dividere in: definizione dei requisiti, progettazione, implementazione, test, distribuzione e manutenzione. Il tempo di produzione equivale alla somma delle singole

fasi. Possiamo notare come avere una documentazione adeguata sia molto importante in quanto tornare indietro tra una fase e l'altra è tutt'altro che semplice. Questo si può vedere nella fase di testing che, come possiamo notare dalla figura 2.1, comincerà subito dopo la fase di progettazione. Questo potrebbe portare a difficoltà se riscontrassimo errori a valle del codice, i quali potrebbero influenzare tutto ciò che è stato sviluppato successivamente.

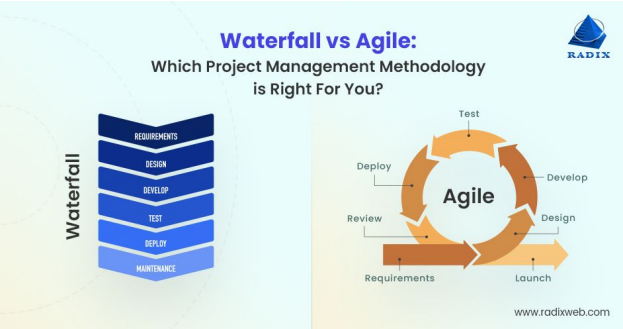

Figura 2.1: Waterfall vs Agile [15]

La differenza principale tra i due metodi [3] risiede nel fatto che, mentre nel primo caso ogni fase può essere iniziata solo quando la successiva è terminata, nel secondo separiamo il prodotto in tante piccole parti che vengono sviluppate separatamente. Proprio grazie a questa caratteristica Agile permette di diminuire il tempo di sviluppo ed è possibile già avere dopo poco tempo un prodotto parziale ma funzionante. In questo caso avremo anche fasi di test continue che permettono di tenere sotto controllo la mantenibilità e la correttezza del codice. D'altra parte, per mantenere sempre costante questo sviluppo c'è bisogno di un continuo contatto tra il cliente e il fornitore, che dovrà adattarsi alle nuove richieste e alle modifiche in itinere.

2.2 Scelte architetturali

Nel settore bancassurance il processo di vendita di una polizza segue un insieme di stati ben definiti che seguono precise regole di transazione. Per gestire il flusso operativo di vendita di una polizza è stato scelto di adottare un'architettura basata su una State Machine in tal modo da trattare il ciclo di vita come una sequenza di stati isolati tra loro. Grazie a questa macchina a stati è possibile coordinare in modo ottimale la vendita del nostro prodotto, separare ognuna delle fasi e gestire facilmente le corrette transizioni tra i vari stati.

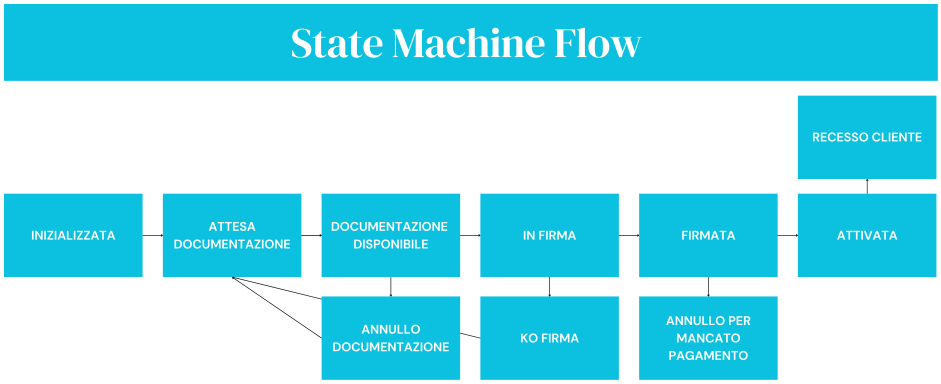

Figura 2.2: State Machine Flow

Ogni stato della State Machine corrisponde ad una precisa fase di vendita del nostro prodotto. Andiamo a vedere nel dettaglio il flow del processo:

INIZIALIZZATA La polizza è stata associata ad una State Machine, in questa fase abbiamo avviato il processo di acquisto della polizza e possiamo iniziare a fornire i dati necessari nei vari campi. Una volta compilati tutti i questionari ed aver fornito le generalità richieste, riceviamo i dati definitivi sul contratto di acquisto della polizza. Dopo averli consultati il sistema procederà con la generazione dei documenti.

ATTESA DOCUMENTAZIONE In questa fase il sistema produrrà un serie di documenti associati sia alle informazioni fornite precedentemente che alla situazione attuale dell'utente. Questi documenti sono generati ogni volta con chiavi diverse. Nel caso il processo fallisca dovranno essere nuovamente generati.

DOCUMENTAZIONE DISPONIBILE Tutti i documenti sono pronti per essere consultati e l'utente può procedere a scaricarli e prenderne visione.

ANNULLO DOCUMENTAZIONE Ci sono stati errori con la generazione di documenti o con l'accesso a qualcuno di questi. Bisogna tornare alla fase precedente e rigenerare nuovamente tutta la documentazione.

IN FIRMA In questa fase l'utente, tramite firma digitale, provvederà a firmare la presa visione dei documenti precedentemente consultati e le clausole del contratto di polizza. Come la documentazione queste fasi sono le più critiche del flow e vengono gestite in modo asincrono con coreografie apposite.

KO FIRMA Ci sono stati errori durante la fase di firma e bisogna tornare indietro e generare nuovamente i documenti da firmare.

FIRMATA L'esito delle firme è andato a buon fine. Viene visualizzata la pagina per scegliere il metodo di pagamento. La polizza rimarrà sospesa fino a quando non verrà accertato il pagamento stesso.

ANNULLO MANCATO PAGAMENTO In caso di errori o mancato pagamento viene annullato tutto il flow.

ATTIVATA La polizza è stata attivata ed è consultabile per il post vendita.

RECESSO CLIENTE Oltre a consultare i dati, il cliente successivamente alla vendita può decidere di recedere.

2.3 Architettura a Microservizi

In ambito bancario-assicurativo l'esigenza di avere maggiore scalabilità e continua evoluzione tecnologica ha portato le aziende di questo settore a migrare dalle precedenti applicazioni centralizzate, formate da blocchi di codice fortemente connessi, a sistemi sempre più piccoli e decentralizzati. Il primo passo verso l'architettura modulare è stato l'utilizzo di SOA (Service Oriented Architecture). Con SOA non abbiamo più un unico blocco di codice, ma più servizi che comunicano tra loro. Questi servizi sono caratterizzati da interfacce comuni, in tal modo che possono essere facilmente incorporati in nuove applicazioni. Per gestire la comunicazione tra di essi viene utilizzato un componente software chiamato ESB (Enterprise Service Bus). Questo elemento però introduce un single point of failure che non ci permetterebbe di ottenere l'indipendenza totale tra i vari servizi. Così negli ultimi anni si è deciso di virare verso l'architettura basata sui microservizi[5] [17] [11].

2.4 Vantaggi e svantaggi

L'architettura a microservizi permette di sviluppare un'applicazione come un insieme di piccoli e autonomi servizi che comunicano tra di loro attraverso protocolli leggeri. Ogni modulo può essere sviluppato in maniera indipendente con linguaggi e tecnologie differenti. I protocolli di comunicazione maggiormente usati sono API Rest, SOAP, gRPC oppure comunicazione asincrona (RabbitMQ oppure Kafka).

Vantaggi

- **Indipendenza**: Posso modificare ogni microservizio senza impattare sugli altri.
- **Mantenibilità**: Il codice è più semplice da comprendere e mantenere, essendo diviso in unità più piccole.
- **Scalabilità**: I microservizi più utilizzati possono essere scalati facilmente senza dover scalare l'intera applicazione.
- **Resistenza**: Anche se un microservizio diventa indisponibile, il resto dell'applicazione continua a funzionare.

Svantaggi

- **Complessità**: Con molti microservizi, può essere complicato gestire il monitoraggio, il logging e il deployment.
- **Comunicazione distribuita**: I problemi di rete e i tempi di comunicazione possono influire negativamente sulle performance.
- **Overhead**: Ogni microservizio richiede la propria infrastruttura, come database, container, ecc., aumentando i costi operativi.

Lo sviluppo di tecnologie come Docker e Kubernetes ha aiutato la crescita esponenziale dell'utilizzo di questa architettura nell'ultimo decennio. Il concetto di container, infatti, supporta perfettamente la divisione del lavoro in moduli a sé stanti. Le tecnologie di Kubernetes ci permettono invece di gestire il service discovery e il load balancing, che sono molto importanti per mantenere in vita un'applicazione. L'utilizzo di microservizi va di pari passo anche con le pratiche di sviluppo chiamate DevOps, che si basano su CI/CD.

2.5 Pattern architetturali

2.5.1 Sincrono vs asincrono nei microservizi

Nelle applicazioni a microservizi, la comunicazione tra i diversi moduli è uno degli aspetti più importanti perché influenzerà le prestazioni e la scalabilità dell'intero sistema. Ci sono due possibilità per implementare la comunicazione: sincrona e asincrona. Ognuna delle due scelte ha le proprie peculiarità

e i propri svantaggi, proprio per questo per scegliere di quale abbiamo bisogno nella nostra architettura dobbiamo analizzare cosa vogliamo ottenere e a cosa possiamo rinunciare. Nel nostro caso abbiamo deciso di usarle alternativamente per avere un'applicazione che usi i punti di forza di entrambe [5].

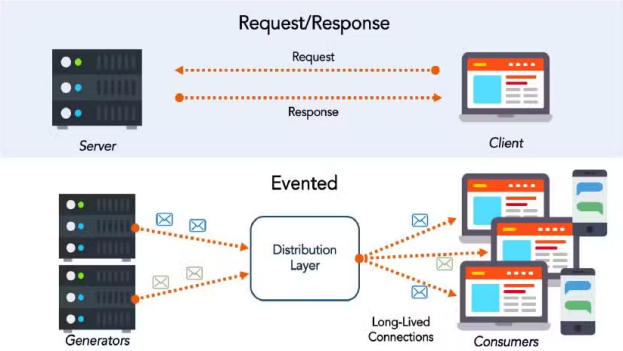

Figura 2.3: Sync vs Async communication [9]

2.5.2 Comunicazione sincrona

La comunicazione sincrona si ha quando un microservizio invia una richiesta a un altro e rimane in attesa fino a quando la risposta è disponibile. Questo comportamento è tipico nell'API RESTful che utilizzano il protocollo HTTP per scambiare dati. I principali vantaggi di questo approccio sono la semplicità di sviluppo e di ricerca degli errori, in quanto il protocollo REST è molto intuitivo e ben supportato dalla comunità. Anche la ricerca di eventuali errori di comunicazione è semplificata, dato che le chiamate avvengono in modo consecutivo e si può risalire facilmente al servizio irraggiungibile o che non ha risposto come ci aspettavamo. Questo stretto collegamento tra i microservizi però fornisce anche i maggiori svantaggi di questo pattern. Se uno qualsiasi dei moduli che vengono chiamati fosse, per esempio, offline o impiegasse troppo tempo a rispondere, l'intera catena si interromperà non tornando una risposta utilizzabile. Anche un elevato numero di richieste potrebbe sovraccaricare uno dei moduli e bloccare l'intero processo.

2.5.3 Comunicazione asincrona

La comunicazione asincrona si adatta bene allo sviluppo di un sistema distribuito. Elimina la necessità di attendere una risposta, in modo da non

collegare strettamente l'esecuzione di due o più servizi. Ci sono diversi metodi per implementare la comunicazione asincrona. Si possono usare code di messaggi, come RabbitMQ, oppure log distribuiti, come Kafka oppure architetture Event Driven. I migliori vantaggi li otteniamo in termini di scalabilità. Non contattando direttamente un servizio, se esso risulta offline, riceverà il messaggio, che non verrà perso, solo quando tornerà disponibile. Così evitiamo anche sovraccarichi del microservizio stesso ma anche del chiamante, in quanto non è vincolato ad aspettare una risposta dall'esterno. La complessità di sviluppo di questo sistema può essere il principale ostacolo. Bisogna gestire code di messaggi e rollback in caso di errori. Anche la disponibilità dei dati deve essere sempre coerente, altrimenti si potrebbero consultare dati non aggiornati in tempo reale e, ultimo, ma non meno importante, il bus potrebbe diventare un punto di fallimento dell'intero sistema.

Capitolo 3

Implementazione del progetto

3.1 Descrizione dell'architettura generale

Il progetto segue un'architettura a microservizi. Il nostro ruolo è stato quello di creare i servizi necessari per la vendita di una polizza. Per portare a buon fine questo processo dovremo interagire anche con servizi esterni di altre società che contemporaneamente a noi lavorano per l'azienda appaltatrice. Nel nostro caso abbiamo deciso di usare più microservizi: Front End, Front End 2 Back End (chiamato `FE2BE`), Back End e Libraries. Questi moduli interagiranno tra loro per fornire un'esperienza ottimale al cliente e agli operatori. Nel Front End sono stati gestiti i quattro canali di vendita su cui si potrà operare: web application, mobile application, desktop application e l'ultimo canale dedicato agli operatori della banca. Ogni Front End potrà scambiare i dati con il proprio FE2BE. Il Front End 2 Back End riceve le chiamate in entrata dal Front End, andando a fare controlli anche sulla sessione attiva, e si occupa soprattutto del flow di vendita e della gestione del post-vendita. Nel Back End troveremo sia le API che gestiranno le transizioni tra gli stati della State Machine, sia quelle che provvederanno ad interagire con altri servizi per acquisire dati sulla polizza. Inoltre, saranno disponibili API esposte per permettere l'integrazione con altri microservizi, fornendo accesso ai dati e alle funzionalità necessarie. Le Libraries forniranno componenti riutilizzabili come classi, interfacce e metodi che supportano e definiscono le funzionalità dei vari moduli dell'applicazione, promuovendo la riusabilità e l'organizzazione del codice.

3.1.1 Front End

Accenni a struttura Front End

Il Front End verrà illustrato soltanto per completezza di progetto in quanto è stato sviluppato da un altro team interno alla nostra società, con il quale abbiamo interagito continuamente per accordarci sulle metodologie di sviluppo e i dati da scambiarci. Sono stati realizzati diversi moduli per ogni possibile canale, così una volta ricevuti i dati dal `FE2BE`, verranno visualizzati nel modo più consono a seconda dell'applicazione che si sta utilizzando. Nelle prossime immagini potremo vedere il flusso di pagine che permettono di acquistare una polizza partendo da un credito precedentemente erogato.

Mock di esempio

Per acquistare il prodotto di polizza entreremo nella sezione dedicata nella quale possiamo selezionare le coperture che vogliamo integrare nella polizza:

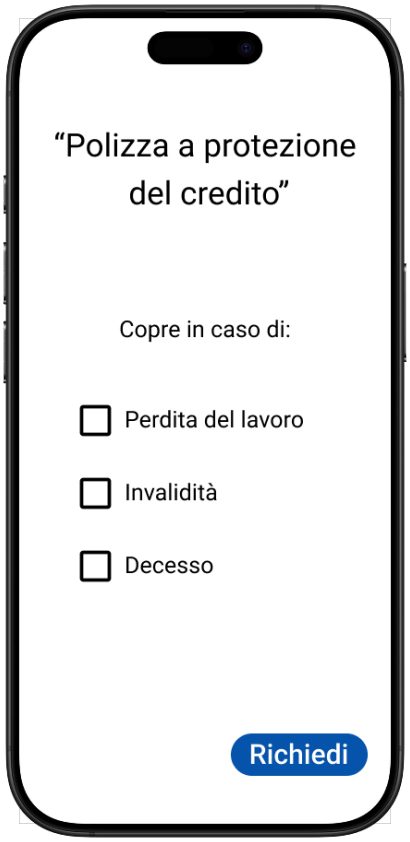

Figura 3.1: Polizza

Il primo passo consiste nello scegliere il credito a cui vogliamo aggiungere il prodotto "Polizza a protezione del credito".

Figura 3.2: Lista crediti

Una volta scelto partirà il nostro flusso di vendita. Il primo passaggio consiste nel rispondere ai questionari sulla situazione patrimoniale, di salute e familiare.

Figura 3.3: Questionari salute, famiglia e lavoro

Una volta compilati e salvati tutti i dati necessari otterremo una proposta di polizza valida, così potremo consultare il riepilogo delle informazioni fornite e confermare la richiesta di attivazione.

Figura 3.4: Riepilogo

Prima di procedere con l'acquisto bisogna firmare i documenti tramite firma digitale, un sistema sviluppato dalla banca in precedenza che gestisce autonomamente tutti i processi necessari per completare con successo il processo di firma.

Figura 3.5: Pagina Documenti e Firma

Terminata la vendita, l'utente vedrà la schermate di operazione andata a buon fine e successivamente potrà accedere alla sezione polizze per consultare il prodotto appena acquistato. Nel caso di errori ci saranno pagine dedicate che segnaleranno il tipo di errore che si è verificato, indicando una possibile soluzione o la pagina da cui ripartire.

Figura 3.6: Pagina di successo

Figura 3.7: Pagina di errore

3.1.2 Front End 2 Back End

Questo modulo funge da tramite per la comunicazione tra il Front End e il Back End. Il primo scopo di questo modulo è quello di verificare la sessione ricevuta e controllare se corrisponde all'utente attivo in quel momento. Le API presenti in questo modulo si occupano anche della creazione di pagine inerenti al flusso di vendita e gestione del post vendita del prodotto. E' possibile ottenere le informazioni necessarie grazie a un framework di terze parti, utilizzato dal sistema, che fornisce un meccanismo configurabile per la gestione delle proposte di polizza. Tale microservizio consente di definire un prodotto assicurativo e, una volta configurato, di supportare l'intero ciclo di vita di una proposta: inizializzazione, salvataggio dei dati e quotazione. Inoltre, il framework permette di modellare un flusso di pagine associato alla proposta stessa. Attraverso specifiche API, getPage e saveData, il Front End può interrogare il Back End per ottenere la struttura e gli elementi dinamici delle pagine, garantendo un'esperienza utente coerente con la configurazione definita. In questo modello, il Front End non contiene logiche di presentazione predefinite, ma si adatta dinamicamente ai dati ricevuti, risultando flessibile ed estensibile a diverse configurazioni di prodotto senza la necessità di modifiche al codice client.

SessionCheck

Per garantire che ogni utente possa accedere solo ai propri dati e non a quelli di altri, è stata aggiunta la funzione (3.1), che verifica la sessione in ogni chiamata e assicura il rispetto delle autorizzazioni.

```
1  SessionData
2  {
3      private String codFiscale;
4      private String userId;
5  }
6
7  private void checkSession(String policyNumber) throws
       Exception {
8      SessionData sessionData = Mappers.getMapper(SessionData.
           class).mapSession(getSession());
9
10     Policy policy = getPolicyFromId(policyNumber);
11
12     String userId_session = sessionData.getUserId();
13     String userId_policy = policy.getUserData().getId();
14
15     if (userId_policy==null || !userId_policy.equals(
           userId_session))
```

```
16          throw new RuntimeErrorException(PolicyError.
                UNAUTHORIZED);
17  }
```

Listing 3.1: Session Check

Dopo aver verificato, per ogni chiamata API, che l'identità dell'utente corrisponda alla sessione attualmente attiva nel browser, sarà possibile ricevere una risposta. In caso contrario, verrà restituito un errore `UNAUTHORIZED`, che indica la mancanza delle autorizzazioni necessarie per eseguire azioni su questa polizza.
Come accennato in precedenza, una funzione importante di questo modulo è la gestione del flusso di vendita del prodotto, permettendo la comunicazione tra Front End e Back End. Le funzioni `getPage` e `saveData` svolgono un ruolo chiave nell'abilitare e supportare questo processo. La **getPage** viene usata dal Front End per ricevere la giusta pagina del flusso da visualizzare all'utente, la **saveData**, invece, permette di salvare in un oggetto i dati che l'utente ha inserito in quella pagina.

GetPage

La nostra API `getPage` (3.2) ha il compito di restituire le informazioni necessarie per la visualizzazione di una pagina all'interno del flusso di vendita. Questa API riceve in input tre parametri:

- **pageId**: identifica la pagina da recuperare in base al flusso attuale.
- **policyId**: identifica univocamente la polizza associata all'operazione in corso.
- **fromPageId**: indica la pagina precedente nel flusso di navigazione, garantendo la coerenza del percorso dell'utente.

Prima di procedere, la funzione esegue un controllo di sessione sulla polizza tramite `checkSession(policyId)` per garantire che l'utente abbia i permessi necessari. L'oggetto restituito `GetPageResource` contiene:

- una mappa di titoli della pagina;
- una lista di pulsanti disponibili;
- una lista di frame per la struttura dell'interfaccia;
- informazioni aggiuntive utili alla visualizzazione.

Infine, il servizio risponde con un `ResponseEntity` contenente la risorsa e lo stato HTTP 200 (OK).

```
public class GetPageResource{
    private Map<String, String> title;
    private List<Button> buttons;
    private List<Frame> frames;
    private Map<String, Object> additionalInfo;
}
public ResponseEntity<GetPageResource> getPage(@PathVariable
   String pageId, @PathVariable String policyId,
   @RequestParam String fromPageId) throws Exception {

    checkSession(policyId);

    GetPageInput getPageInput = getPageFactory.create(pageId,
        policyId, fromPageId);
    GetPageCommand getPageCommand = beanFactory.getBean(
       GetPageCommand.class, getPageInput);
    GetPageOutput getPageOutput = getPageFlowCommand.execute
       ();
    GetPageAssembler getPageFlowAssembler = new
       GetPageAssembler(SaleController.class);
    GetPageResource getPageResource = getPageAssembler.
       toModel(getPageOutput);

    return new ResponseEntity<>(getPageResource, HttpStatus.
       OK);
}
```

Listing 3.2: Get Page

SaveData

Ricevuta la pagina di interesse si può procedere con la compilazione dei campi richiesti attraverso la **saveData** (3.3). Questa API verrà invocata ogni qual volta l'utente cambia pagina del flusso, andando a salvare i valori inseriti nei vari form, questionari o campi di testo. L'API riceve in input i seguenti parametri:

- **policyId**: identifica univocamente la polizza associata al processo di vendita.
- **elements**: una mappa contenente i dati inseriti dall'utente, strutturata in chiave-valore.
- **pageId**: identifica la pagina corrente del flusso di navigazione.

La funzione ci restituisce un oggetto di tipo `DataObjectResource` che è costituito da una lista di elementi, definiti dalla classe `Element`. Ogni `Element` contiene:

- **key**: chiave univoca dell'elemento salvato.
- **value**: valore inserito dall'utente.
- **design**: struttura e tipologia dell'elemento (form, campo di testo, questionario).

Questa API garantisce il corretto salvataggio dei dati nel flusso di vendita, mantenendo la coerenza delle informazioni durante la navigazione dell'utente.

```
1  public class Element {
2      private String key;
3      private String value;
4      private Design design;
5
6  }
7  public class DataObjectResource{
8      private List<Element> elements;
9  }
10
11 public ResponseEntity<DataObjectResource> saveDataFields(
       @PathVariable String policyId, @RequestBody Map<String,
       Object> elements, @RequestParam String pageId) throws
       Exception {
12
13     checkSession(policyId);
14
15     DataObjectInput dataObjectInput = dataObjectFactory.
           create(policyId, elements, pageId);
16     DataObjectCommand dataObjectCommand = beanFactory.getBean
           (DataObjectCommand.class, dataObjectInput);
17     DataObjectOutput dataObjectOutput = dataObjectCommand.
           execute();
18     DataObjectAssemblerdataObjectAssembler = new
           DataObjectAssembler(SaleController.class);
19     DataObjectResource dataObjectResource =
           dataObjectAssembler.toModel(dataObjectOutput);
20
21     return new ResponseEntity<>(dataObjectResource,
           HttpStatus.OK);
22 }
```

Listing 3.3: Save Data

3.1.3 Back End

Il Back End rappresenta il nucleo principale della nostra applicazione. Si occupa principalmente di gestire la creazione e gli spostamenti della State Machine, di inizializzare la polizza e di chiamare e interagire con il core di microservizi esterni. Le API che andremo ad analizzare nel dettaglio sono la **setupPolicy**, la **policyInfo** e l'**documentPage**. Oltre a queste sono presenti le funzioni che gestiscono la State Machine ma verranno analizzate in seguito in altre sezioni.

SetupPolicy

La `setupPolicy` è una delle principali funzioni del core. Si occupa di inizializzare il flusso di vendita della polizza a partire dal credito originale. Per fare ciò, questa API dovrà recuperare dati da diversi microservizi. Il primo contatto avviene con il microservizio che si occupa dell'anagrafica degli intestatari. In questo caso ci saranno vari controlli sull'intestatario principale e su eventuali cointestatari e verranno recuperati tutti i dati anagrafici disponibili su di loro. Successivamente vengono eseguiti controlli sull'ammissibilità della polizza, verificando i requisiti, lo stato di salute e i rischi degli intestatari. Se tutti questi controlli risultano positivi, il microservizio esterno che si occupa dei crediti inizializza i valori relativi alla polizza. Tra i principali troviamo:

- **Importo della polizza**: L'importo totale da assicurare, determinato in base al credito originale.
- **Premi assicurativi**: Gli importi da pagare per la polizza, che possono variare in base al tipo di copertura e ai rischi associati.
- **Durata della polizza**: Il periodo durante il quale la polizza sarà attiva, che può variare a seconda delle condizioni del contratto.
- **Beneficiari**: Le persone o entità designate a ricevere il risarcimento in caso di sinistro, come previsto dalla polizza.

Questi valori vengono poi utilizzati per completare la `PolicyResource`, che conterrà tutte le informazioni relative alla polizza per ciascun utente.

```
1 public class UserPolicyDetail{
2     private String codFiscale;
3     private double policyAmount;
4     private double insurancePremium;
5     private int policyDuration;
6     private List<String> beneficiaries;
```

```
}

public class PolicyResource{
    private Esito esito;
    private String policyId;
    private List<UserPolicyDetail> userPolicyDetails;
}

public class SetupPolicyDto{
    private User user;
    private List<User> intestatari;
    private Data dataEmission;
    private String loanId;
    ...
}

public ResponseEntity<PolicyResource> setupPolicy(
   @RequestBody SetupPolicyDto setupPolicyDto) throws
   Exception {

    //controlli anagrafica
    AnagraficaInput anagraficaInput = anagraficaFactory.
       create(setupPolicyDto.intestatari);
    AnagraficaCommand anagraficaCommand = beanFactory.getBean
       (AnagraficaCommand.class, anagraficaInput);
    AnagraficaOutput anagraficaOutput = anagraficaCommand.
       execute();

    //controlli ammissibilita'
    AmmissibilitaInput ammissibilitaInput =
       ammisibilitaFactory.create(user, anagraficaOutput,
       loanId, ...);
    AmmissibilitaCommand ammissibilitaCommand = beanFactory.
       getBean(AmmissibilitaCommand.class, ammissibilitaInput
       );
    AmmissibilitaOutput ammissibilitaOutput =
       ammissibilitaCommand.execute();

    //se entrambi i controlli risultano positivi, passiamo
       all'inizializzazione
    InitPolicyInput initPolicyInput = initPolicyFactory.
       create(user, anagraficaOutput, setupPolicyDto,
       ammisibilitaOutput,...);
    InitPolicyCommand initPolicyCommand = beanFactory.getBean
       (InitPolicyCommand.class, initPolicyInput);
    InitPolicyOutput initPolicyOutput =  initPolicyCommand.
       execute();
    InitPolicyAssembler initPolicyAssembler = new
       InitPolicyAssembler(PolicyController.class);
```

```
40      InitPolicyResource initPolicyResource =
            initPolicyAssembler.toModel(initPolicyOutput);
41  }
```

Listing 3.4: SetupPolicy

PolicyInfo

La funzione `policyInfo` (3.5) viene invocata dopo l'inizializzazione per ottenere le proposte di polizza create dalla `setupPolicy` per ogni intestatario del credito. Questa API riceve in ingresso il **policyId**, che rappresenta l'identificativo della polizza. L'oggetto restituito, di tipo `PolicyInfo`, contiene una lista di proposte di polizza per ogni intestatario. Al momento dell'acquisto ogni intestatario completerà il proprio processo (firmando i documenti a lui relativi), e solo una volta che tutti avranno completato e firmato la polizza essa andrà in stato "ATTIVATA". Successivamente ogni utente, in base al proprio ruolo (intestatario principale o cointestatario), avrà la possibilità di saldare il proprio premio assicurativo separatamente dagli altri.

```
1  public class PolicyInfo{
2      private List<PolicyProposal> policyProposals;
3  }
4   public ResponseEntity<PolicyInfo> policyData(@PathVariable
       String policyId) throws Exception {
5
6      PolicyInfo policyDataInput = policyInfoFactory.create(
           policyId);
7      PolicyInfoCommand policyInfoCommand = beanFactory.getBean
           (PolicyInfoCommand.class, policyInfoInput);
8      PolicyInfoOutput policyInfoOutput = policyInfoCommand.
           execute();
9
10     PolicyInfoAssembler policyInfoAssembler = new
           PolicyInfoAssembler(PolicyController.class);
11     PolicyInfoResource policyInfoResource =
           policyInfoAssembler.toModel(policyInfoOutput);
12
13     return new ResponseEntity<>(policyInfoResource,
           HttpStatus.OK);
14 }
```

Listing 3.5: PolicyInfo

DocumentPage

L'API `getDocumentPage` viene utilizzata per ottenere informazioni relative ai documenti associati a una polizza, identificata dal parametro **policyId**. La

funzione inizialmente recupera la lista dei documenti associati alla polizza e li assegna alla struttura `ListDocument`, che include sia la lista stessa che l'esito del recupero. Questa lista viene passata al `DocumentPageAssembler` il quale costruisce un oggetto `DocumentPageResource`. Questo oggetto, restituito dall'API, fornisce, al servizio da cui è stato chiamato, un modello strutturato che consente di visualizzare i documenti separati in sezioni, a seconda del tipo di documento, e dettagliati con informazioni legali e note informative.

```
1  public class ListDocument {
2      private List<Document> documents;
3      private String esito;
4  }
5
6  public class DocumentPageResource{
7      private Product productType;
8      private List<String> informative;
9      private List<Section> sections;
10 }
11
12 public ResponseEntity<DocumentPageResource> getDocumentPage(
       @PathVariable String policyId) throws Exception {
13
14     DocumentPageCommand documentPageCommand = beanFactory.
           getBean(DocumentPageCommand.class, policyId);
15     ListDocument listDocument = documentPageCommand.execute()
           ;
16     DocumentPageAssembler documentPageAssembler = new
           DocumentPageAssembler(PolicyController.class);
17     DocumentPageResource documentPageResource =
           documentPageAssembler.toModel(listDocument);
18
19     return new ResponseEntity<>(documentPageResource,
           HttpStatus.OK);
20 }
```

Listing 3.6: DocumentPage

3.1.4 Libraries

Nel nostro progetto, l'architettura del software è supportata da un insieme di Libraries, che supportano e completano i vari moduli dei microservizi. Ognuno di essi infatti ha la propria libreria che permette di migliorare la modularità e la manutenibilità del codice. Esse contengono le strutture dati senza curarsi delle comunicazioni tra i componenti, ottimizzando così l'uso della memoria evitando duplicazioni. Uno dei compiti principali delle librerie è la gestione delle eccezioni. Riprendendo l'esempio della chiamata al servizio

di ammissibilità, se vengono passati valori errati o mancanti, il sistema restituisce un errore che viene gestito tramite una classe dedicata all'interno della libreria. Ad esempio, se la richiesta non contiene un corpo valido o manca l'utente da controllare, vengono generati specifici errori, come mostrato nel seguente codice:

```
public enum AmmissibilitaError {

    NO_REQUEST_BODY("NO_REQUEST_BODY", "Request body missing"
        , HttpStatus.BAD_REQUEST),
    NO_USER("NO_USER", "User missing", HttpStatus.BAD_REQUEST
        );

    private final String code;
    private final String message;
    private final HttpStatus httpStatus;

    AmmissibilitaError(String code, String message,
        HttpStatus httpStatus) {
        this.code = code;
        this.message = message;
        this.httpStatus = httpStatus;
    }

    @Override
    public String getCode() {
        return code;
    }

    @Override
    public String getMessage() {
        return message;
    }

    @Override
    public HttpStatus getHttpStatus() {
        return httpStatus;
    }
}
```

Listing 3.7: ErrorHandling

L'uso di queste librerie semplifica lo sviluppo e la manutenzione del sistema, permettendo agli sviluppatori di concentrarsi sulla logica applicativa senza doversi preoccupare dei dettagli legati alla comunicazione tra i componenti.

3.2 Protocolli di comunicazione

3.2.1 REST API

Introduzione

Le API REST rappresentano una delle più utilizzate architetture per la comunicazione tra microservizi. Un'API è un'interfaccia di programmazione che stabilisce un modo da seguire per comunicare con altri servizi. I servizi più semplici che possiamo immaginare sono un client e un server. Il Client per esempio vuole accedere a delle informazioni presenti online, così invia una richiesta tramite API ed avrà una risposta dal server, che grazie a queste interfacce può condividere in modo sicuro e controllato le risorse di cui dispone. REST(*Representational State Transfer*) invece è un'architettura software che detta alcune regole sul funzionamento delle API stesse. Questa architettura si è diffusa enormemente negli ultimi anni grazie alla sua semplicità e interoperabilità tra sistemi diversi. Si basa sull'uso di protocolli HTTP standard e sulle operazioni CRUD.

Principi Architetturali

Il modello REST si basa su sei principi fondamentali:

- **Interfaccia uniforme**: utilizzo di risorse con URL chiari e metodi HTTP standard.
- **Statelessness**: ogni richiesta HTTP è indipendente e non mantiene stato sul server.
- **Cacheability**: le risposte possono essere memorizzate nella cache per ottimizzare le performance.
- **Sistemi a livelli**: possibilità di inserire livelli intermedi che forniscano supporto come sicurezza e autenticazione.
- **Codice on Demand**: il server può inviare codice eseguibile al client (per esempio durante la compilazione di moduli).

Grazie a questi principi riusciamo ad ottenere notevoli vantaggi nella comunicazione tra microservizi. Grazie alle chiamate stateless e ad un sistema di cache ben gestito riusciamo ad alleggerire il server, il quale non deve mantenere le vecchie chiamate o informazioni, mantenendo la scalabilità e ottime prestazioni. Otteniamo anche flessibilità di sviluppo potendo separare la produzione tra client e server. Una modifica di uno dei sistemi non va

a ripercuotersi sull'altro. Architetture con maggiori livelli permettono una maggiore separazione dei ruoli. Grazie a questa separazione ogni sistema potrà anche essere sviluppato con linguaggi e tecnologie differenti garantendo l'indipendenza tra di essi. Alcuni svantaggi possono presentarsi con problemi dovuti alla latenza e all'uso di chiamate sincrone.

Implementazione

La comunicazione attraverso REST API [18] avviene tramite richieste inviate da un client, che si mette in attesa di ricevere una risposta dal server. Per contattare nel modo giusto il server, il client deve mandare una precisa richiesta consultando la documentazione fornita dalla controparte. La documentazione è molto importante in quanto spiega in modo dettagliato tutti i parametri che un servizio si aspetta e in come restituisce le proprie risorse. Le risposte vengono identificate da codici di stato comuni al mondo del web. Una richiesta è formata da: URL, Metodo, Headers HTTP, risorsa e parametri. L'URL indica l'indirizzo univoco a cui vogliamo accedere per ottenere i dati e permette al server di capire ciò di cui abbiamo bisogno. Il metodo HTTP indica l'azione che vogliamo compiere sulla risorsa in questione. Ci sono quattro metodi principali: GET, POST, PUT e DELETE.

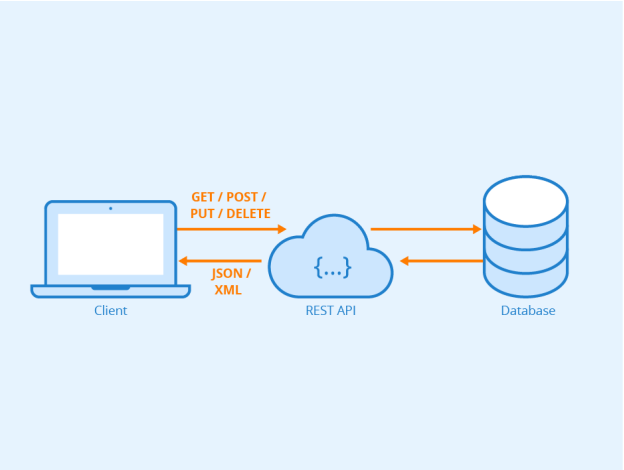

Figura 3.8: Rest Api [7]

Con GET chiediamo di accedere alle risorse disponibili e possiamo completare la richiesta con parametri per filtrare i risultati. POST e PUT sono utilizzati per inviare dati al server remoto. In questo caso dovrà essere riempito anche il campo risorsa della richiesta HTTP. Il primo viene usato maggiormente per creare nuove entità, invece il secondo per aggiornare entità già esistenti

in quanto a differenza del primo è idempotente, cioè inviando più richieste avremo sempre lo stesso risultato. La DELETE viene utilizzata per rimuovere una precisa risorsa. Gli header invece sono usati per scambiare metadati tra i due servizi. Molto utili per tenere traccia di autorizzazioni e sessioni attive per garantire l'autenticità del chiamante. I parametri invece sono utili per completare una richiesta e fornire maggiori dettagli sulla risorsa cui vogliamo accedere. Una volta ricevuta la richiesta il server, dopo averla elaborata, rimanderà una risposta al chiamante. Queste risposte sono formate da: codice di stato, corpo del messaggio e header. Gli header, come nel caso precedente, sono utilizzati per scambiare metadati tra le parti. Il codice di stato è un codice standardizzato, utilizzato nel mondo del web, che permette di riconoscere il tipo di risposta dal codice stesso. Per standard si utilizzano 2XX per indicare richieste a buon fine, 4XX per indicare errori di richiesta o di raggiungimento e 5XX errori interni ad un servizio. I più famosi sono: 200 OK, 403 UNAUTHORIZED, 404 NOT FOUND e 500 INTERNAL SERVER ERROR. Nel corpo del messaggio invece possiamo trovare in JSON o XML la risorsa a cui abbiamo chiesto l'accesso.

API Gateway

Il gateway API è un microservizio che si occupa di gestire le richieste REST che arrivano dall'architettura. Può fungere da orchestratore per garantire autenticazione, permettendo l'instradamento solo alle richieste autorizzate, bilanciamento delle richieste e logging di controllo. All'interno del nostro prodotto possiamo considerare il pattern Front End 2 Back End (F2B) è una variante del classico API Gateway in cui esiste un Back End specifico per ogni tipologia di client che abbiamo implementato (web, app mobile, app desktop e app filiale). Questo approccio riduce il numero di chiamate necessarie ai microservizi e migliora la gestione dei dati lato Front End, soprattutto quando abbiamo bisogno di ricevere dati diversi e risposte ottimizzate in base al client che stiamo utilizzando.

3.2.2 SOAP API

SOAP (Simple Object Access Protocol) è un servizio di messaggistica che usa messaggi di tipo XML per scambiare informazioni tra i servizi. Si è sviluppato contemporaneamente alle service oriented architecture e quindi è più vecchio rispetto alla comunicazione REST. Nonostante la grande diffusione della comunicazione REST, questo servizio rimane ancora usato in molti sistemi distribuiti. Un messaggio SOAP è formato da: header e body. Viene formattato nello stesso modo sia quando è utilizzato come richiesta sia

come risposta. Negli header, come nel caso precedente, possiamo aggiungere informazioni per gestire autenticazione e autorizzazione. Nel body inseriremo dati in formato XML. Come questi messaggi potranno essere scambiati a livello trasporto (del modello OSI) dipende dal SOAP Binding che decidiamo di usare. Possiamo usare sia RPC che Document Style. RPC viene usato per scambiare messaggi più piccoli e semplici e si basa su un classico scambio sincrono di messaggi tra client e server. Il messaggio è incapsulato dentro un SOAP envelope che contiene anche due header per specificare il tipo di messaggio (Content-Type) e la lunghezza del contenuto (Content-Length). Invece, Document Style usa messaggi più complessi e orientati verso la comunicazione message-oriented.

3.2.3 Confronto REST vs SOAP

Adesso andremo ad analizzare le performance, guardando le analisi effettuate dalla ricerca [19], di entrambi i protocolli per spiegare le scelte prese nel progetto. Il protocollo REST è più facile da implementare utilizzando i metodi HTTP standard invece che i file WSDL di SOAP. Proprio per questo inserire i dati in un URI è molto più leggero rispetto a un XML Envelope. Ed è proprio questo payload corposo che rallenta le operazioni di scambio con SOAP. In REST, quando abbiamo bisogno di passare un body, utilizzeremo un formato più leggero e semplice come il JSON. Come possiamo notare dal grafico sottostante, in media una chiamata REST impiega 7 ms in meno. Oltre a occupare più memoria anche la CPU ne risente in tempi di elaborazione.

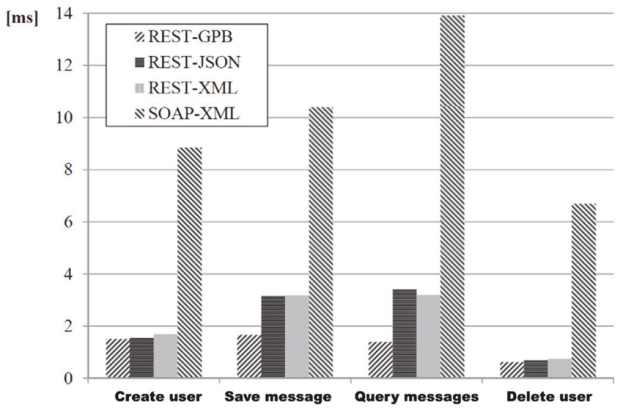

Figura 3.9: Response Time comparison [16]

Anche a livello di gestione degli errori è preferibile usare REST in quanto fornisce un retry automatico che non va implementato lato client. Dal lato

della sicurezza invece possiamo affermare che SOAP è più sicuro grazie ad un avanzato sistema di crittografia e autenticazione anche se, negli ultimi anni, il protocollo HTTPS può essere migliorato con accorgimenti ed estensioni. Proprio grazie alla sua affidabilità il protocollo SOAP è ancora utilizzato in settori in cui la sicurezza svolge un ruolo fondamentale, come il nostro settore bancario. Come vedremo successivamente infatti, anche se abbiamo preferito migrare la maggior parte dei microservizi verso il paradigma REST utilizzeremo e andremo a chiamare ancora alcuni servizi di tipo SOAP, soprattutto in ambito del salvataggio e consultazione documenti. Possiamo per esempio vedere le differenze tra JSON e XML utilizzati nei due servizi (3.8) (3.9).

```
POST /api/uploadDocument HTTP/1.1
Content-Type: application/json
Authorization: Bearer <token>
Accept: application/json

{
    "insert": {
        "insertDoc": {
          "docOperation": "docUpload",
          "documento": {
            "document_name": "doc99999",
            "doc_type": "docRiconoscimento",
            "format": "PDF",
            "content": "<!-- contenuto in byteArray -->",
            "attributes": [
              {
                "attribute_name": "cod_fiscale",
                "attribute_values": "XXXYYY99X99Y999X"
              },
              {
                "attribute_name": "cod_documento",
                "attribute_values": "CIE"
              },
              {
                "attribute_name": "data_rilascio",
                "attribute_values": "10/05/2016 00:00:00"
              }
            ]
          }
        }
    }
}
```

Listing 3.8: JSON Message

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soap:Body>
      <insert>
         <insertDoc xmlns="">
            <docOperation>docUpload</docOperation>
            <documento>
               <document_name>doc99999</document_name>
               <doc_type>docRiconoscimento</doc_type>
               <format>PDF</format>
               <content>
               <!-- contenuto in byteArray -->
               </content>
               <attributes>
                  <attribute_name>cod_fiscale</attribute_name
                     >
                  <attribute_values>XXXYYY99X99Y999X</
                     attribute_values>
               </attributes>
               <attributes>
                  <attribute_name>cod_documento</
                     attribute_name>
                  <attribute_values>CIE</attribute_values>
               </attributes>
               <attributes>
                  <attribute_name>data_rilascio</
                     attribute_name>
                  <attribute_values>10/05/2016 00:00:00</
                     attribute_values>
               </attributes>
            </documento>
         </insertDoc>
      </insert>
   </soap:Body>
</soap:Envelope>
```

Listing 3.9: XML Message

3.2.4 Apache Kafka

Kafka è una piattaforma che si occupa di streaming di eventi per gestire la comunicazione real time di continui flussi di dati. La sua architettura si basa su un modello di Publish/Subscribe. Kafka gestisce una serie di topics sui quali vengono salvati gli eventi che arrivano dai vari microservizi. Garantisce la durabilità dei dati salvando i messaggi, con specifico indice e offset, su memoria per un tempo specifico. E' scalabile orizzontalmente

dato che utilizza cluster che possono essere aggiunti aumentando i server e aumentando la loro capacità. Infine uno dei principali vantaggi di Kafka è la sua capacità di gestire carichi di lavoro variabili e di garantire l'affidabilità dei dati anche in caso di errori hardware o interruzioni di rete replicando le informazioni sui numerosi server di cui fa utilizzo. Gli elementi principali dell'architettura Kafka sono: Producer, Consumer e Topic. Un Producer è un modulo che invia informazioni in un cluster, queste informazioni vengono salvate sotto forma di messaggi e indicizzate per poter essere poi recuperate. Un Consumer è un servizio che legge i dati di un certo Topic dal cluster. Solitamente i dati sono letti in ordine ma si può accedere a dati più vecchi manipolando l'offset. Possono esserci indifferentemente sia più consumer che più producer per uno stesso topic. In questo modo, grazie alla comunicazione asincrona, riusciamo a scollegare completamente tra loro il servizio che invia i dati e il servizio che riceverà il messaggio stesso. I Topic sono log file presenti all'interno di un cluster Kafka. Il consumer si occuperà di gestire l'offset per accedere alla giusta posizione. I Topic possono essere divisi in più partizioni, ma ogni messaggio verrà registrato solo su una di esse. Otteniamo la scalabilità avendo la possibilità di avere queste partizioni su diversi broker dello stesso cluster Kafka.

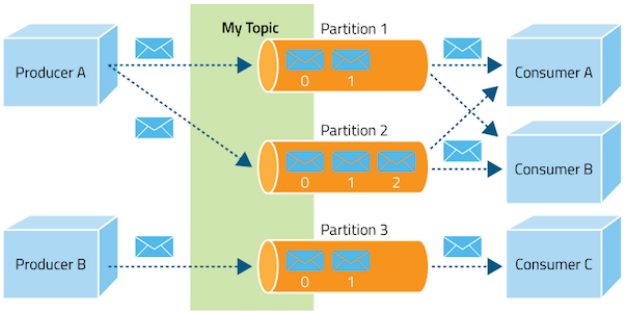

Figura 3.10: Kafka Topic [23]

E' perfettamente integrabile con sistemi come Spring Boot grazie alle funzionalità del Kafka Connect. Permette di scrivere una serie di connettori per integrare il servizio di streaming dati tra servizi di natura diversa. Grazie alla libreria Kafka Stream permette di analizzare grandi quantità di dati e mandare i risultati sia a Kafka stesso oppure a sistemi esterni. Questo stream viene utilizzato per gestire analisi real-time, sistemi di allerta e monitoraggio della sicurezza.

3.3 Uso di Spring

In aiuto per la creazione di applicazioni di grandi dimensioni ci viene dato Spring Framework, un framework open source fornito da Java. Spring, per le sue caratteristiche, è perfetto per la creazione di microservizi [6], grazie alla separazione tra logica di business e configurazione, permettendo di definire componenti e di gestire automaticamente le loro dipendenze. Fornisce diversi moduli che permettono di controllare tutte le fasi di vita di un'applicazione. Per esempio abbiamo Spring Boot per lo sviluppo di applicazioni web, Spring Cloud per la gestione di microservizi distribuiti, Spring Security che gestisce la sicurezza delle API tramite Authorization Token e Spring Data che abilita le interazioni con i database che possono essere sia SQL che NoSQL.

3.3.1 Spring Boot

Spring Boot è il modulo principale del framework [21]. Tramite le sue funzionalità, permette di ridurre i tempi di produzione e facilita le operazioni fornendo configurazioni automatiche per implementare un'applicazione Spring. I principi chiave che caratterizzano Spring sono:

- **Inversion of control**
- **Dependency Injection**
- **Aspect-Oriented Programming**

Inversion of Control
Questo principio permette di invertire il controllo della creazione delle dipendenze, delegandolo al framework invece di gestirlo manualmente nel codice, così da ridurre le connessioni tra le classi.

Dependency Injection
Un meccanismo che consente di fornire le dipendenze richieste da un componente senza che sia necessario istanziarle direttamente.

Aspect-Oriented Programming
Un paradigma che permette di separare le preoccupazioni trasversali (come logging e sicurezza) dal codice applicativo principale.

In Spring, gli oggetti sono creati e collegati insieme da software container che si occupano della dependency injection. Possono esserci container di tipo Bean Factory (utili in casi semplici) oppure ApplicationContext, più complessi e adatti a scenari più grandi. La gestione del ciclo di vita di un Bean è

affidata ad un'interfaccia che supporta la loro creazione, le dipendenze, inizializza i metodi e si occupa dell'autowiring. Invece, d'altro canto, le interfacce per l'ApplicationContext aggiungono la gestione dinamica delle risorse, del publish/subscribe pattern e dell'annidamento del context. Per definire un Bean bisogna marcarlo con un'annotazione (`@Component`). Ogni bean viene individuato, instanziato, cablato e registrato automaticamente all'interno dell'ApplicationContext, purché faccia parte di un pacchetto scansionato. I bean dichiarati come @Configuration hanno bisogno di un'ulteriore ispezione.

3.3.2 Spring Data

Spring Data è il modulo che si occupa della persistenza dei dati in database sia relazionali che non relazionali. Si basa sul concetto di `@Repositories`, interfacce implementate automaticamente dal modulo in oggetto. Ci sono tre possibili alternative per interagire con i DBMS: Spring Data JDBC, Spring Data JPA e Spring Data R2DBC.
Spring Data JDBC è un framework di mapping relazionale a oggetti per database relazionali che mira a evitare la complessità della maggior parte degli altri framework ORM. È l'approccio più semplice per interagire con i database relazionali. Per caricare un'entità viene chiamata la corrispondente query SQL senza ricorrere a lazy loading o caching. Per salvarla bisogna utilizzare esplicitamente il metodo save(...) altrimenti le modifiche verranno perdute, dato che non viene usato il dirty tracking. Le entità sono mappate alle tabelle del database in modo semplice; queste tabelle devono essere create precedentemente in quanto non possono essere dedotte automaticamente dal framework. Vengono usate annotazioni come `@Id`, `@Table` e `@Column` per fornire maggiori dettagli riguardo il mapping [22].
Spring Data JPA è un'estensione di JPA (Jakarta Persistence API), un framework ORM che semplifica l'interazione tra database relazionali. Permette di convertire automaticamente oggetti Java in tabelle del database e viceversa, grazie all'utilizzo di Hibernate, mantenendo costantemente la sincronizzazione tra le due. La comunicazione con il database viene affidata all'Entity Manager che gestisce le connessioni e fornisce tutti i metodi CRUD per interagire con esso, così da risparmiare al programmatore la scrittura delle query necessarie. Per ogni classe annotata con `@Entity` abbiamo di default una corrispettiva tabella nel DB. Per fare delle modifiche dovremo usare annotation come `@Table` che affida al programmatore il compito del mapping. Ogni entità deve avere un campo `@Id` che rappresenta la chiave primaria, che può essere naturale, quando viene assegnata fin dall'inizio, oppure surrogata (`@GeneratedValue`), quando viene attribuita successivamente all'aggiunta della prima entry. Oltre alla chiave primaria viene usata la foreign key per

fare riferimento ad altri schemi. Le relazioni tra tabelle possono essere di vari tipi: `@OneToOne`, `@ManyToOne` oppure `@ManyToMany`.
Infine abbiamo Spring Data R2DBC che è utilizzato nei database reattivi. È ideale per applicazioni non bloccanti che sfruttano WebFlux o altri framework reattivi. Non supporta JPA, in quanto è bloccante, e richiede di gestire il mapping manualmente.

3.3.3 Spring WebMVC

Spring offre due diverse alternative per sviluppare web application: Spring MVC, metodo classico basato su codice bloccante e sulle astrazioni servlet e Spring WebFlux, moderno metodo basato sugli stream reactive. Spring MVC è un design pattern che fornisce una soluzione separando l'applicazione in Model, View e Controller. Il Controller, annotato in Spring Boot con `@Controller`, si occupa delle richieste in entrata che possono arrivare via URL (`@PathVariable`)), via query string (`@RequestParam`) oppure via request body (`@RequestBody`). I dati in ingresso di norma vengono controllati (con il comando `@Valid`) per lanciare un'eccezione nel caso non corrispondessero a ciò che ci aspettavamo. Una volta elaborati i dati di ingresso interagisce con il Model che si aggiorna e restituisce i dati. Nel frattempo, la View genera una risposta basandosi sul Model attuale.

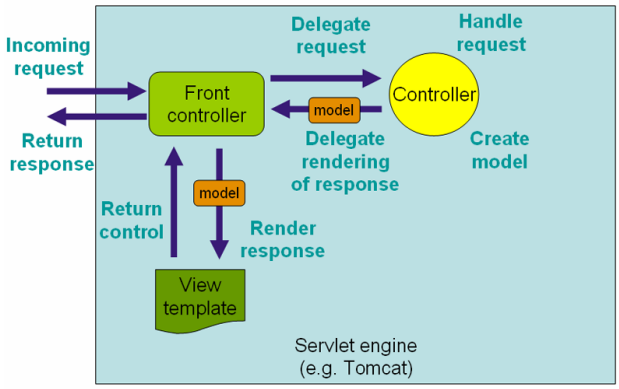

Figura 3.11: Architettura di Spring MVC [8]

3.3.4 Spring Security

Spring Security è un framework che fornisce autenticazione, autorizzazione e protezione per applicazioni Java basate su Spring. È altamente configurabile

e si integra perfettamente con Spring Boot, rendendo sicuro lo sviluppo di applicazioni web e API.

Le principali funzionalità di Spring Security includono:

- **Autenticazione**: Fornisce un metodo affidabile per accertare l'identità degli utenti che accedono al sistema.
- **Autorizzazione**: Applica controlli di autorizzazione adeguati per garantire che gli utenti autenticati possano accedere solo alle risorse e svolgere solo le azioni per cui sono autorizzati.
- **Privacy e integrità dei dati**: Garantisce che i dati scambiati con l'applicazione non possano essere intercettati o modificati da terze parti durante la trasmissione e dopo.
- **Responsabilità**: Fornisce un meccanismo che associa agli utenti le proprie azioni, raccogliendo logging e monitoraggio per garantire il rispetto delle politiche di sicurezza.

Per fare ciò utilizza: filtri `Servlet` che vanno a controllare le richieste HTTP entranti, `Aspect` che vanno a controllare se le richieste provengano dalla giusta entità e promuove protocolli standard per adattarsi al mercato e fornire soluzioni semplici a problemi noti.

3.3.5 Spring Cloud

Spring Cloud è un insieme di strumenti e framework per costruire sistemi distribuiti e microservizi in modo semplice ed efficace. Si integra con Spring Boot e offre soluzioni per:

- **Configurazione centralizzata**
- **Service discovery**
- **Bilanciamento del carico**
- **Gestione delle API Gateway**
- **Circuit breaker** (tolleranza ai guasti)
- **Distributed tracing** (monitoraggio e logging distribuito)

Spring Cloud è ideale per le architetture a microservizi, aiutando gli sviluppatori a superare le sfide tipiche dei sistemi distribuiti.

3.4 Pattern implementati e motivazioni

3.4.1 State Machine

Come pattern per gestire la vendita della polizza è stato scelto di associare a ciascuna di essa una State Machine che gestisca l'avanzamento e il rollback tra le varie fasi di vendita. Dichiariamo gli stati della State Machine in un file JSON come nel codice successivo (3.10). Oltre agli stati dobbiamo elencare tutte le possibili transizioni tra essi, così potremo controllare la validità degli spostamenti e garantire la loro integrità.

```
{
  "states": [
    "INIZIALIZZATA",
    "ATTESA_DOCUMENTAZIONE",
    "DOCUMENTAZIONE_DISPONIBILE",
    "IN_FIRMA",
    "FIRMATA",
    "ATTIVATA",
    "ANNULLO_DOCUMENTAZIONE",
    "KO_FIRMA",
    "ANNULLO_PER_MANCATO_PAGAMENTO",
    "RECESSO_CLIENTE"
  ],
  "transitions": [
    {
      "from": "INIZIALIZZATA",
      "to": "ATTESA_DOCUMENTAZIONE",
      "name": "prepare-documents"
    },
    {
      "from": "DOCUMENTAZIONE_DISPONIBILE",
      "to": "IN FIRMA",
      "name": "sign-documents"
    },
    ...,
    {
      "from": "FIRMATA",
      "to": "ATTIVATA",
      "name": "activate"
    }
   ]
}
```

Listing 3.10: State Machine State

L'uso di una State Machine che mantenga la polizza sempre in uno stato preciso ci aiuta in vari aspetti della programmazione. Abbiamo più controllo sul flow di vendita, identificando sempre uno stato preciso in cui puoi arrivare

solo da determinati stati, grazie alle transizioni dichiarate precedentemente. Anche in fase di testing può aiutare andando a scrivere test mirati che possano controllare il passaggio di fase o azioni che possano svolgersi solo in un determinato stato. La creazione o i movimenti di questa State Machine associata alla polizza verranno effettuati dal nostro Back End usando l'interfaccia `State Machine Service` (3.11).

```
1 public interface StateMachineService {
2     policyStateMachineResource createStateMachine(
          PolicyStateMachineModel policyStateMachineModel);
3     StateMachineResource getStateMachine(String
          stateMachineId);
4     void updateStateMachine(String stateMachineId,
          UpdateStateMachineRequest updateStateMachineRequest);
5     MoveStateMachineOutput moveStateMachine(String
          stateMachineId, String event);
6 }
```

Listing 3.11: State Machine Service

I tre servizi principali servono per creare la State Machine all'inizio del processo, settando lo stato a inizializzato e associandola ad una polizza. Per cambiare lo stato useremo una move che ricevendo il tipo della transizione provvederà al passaggio di stato, dopo aver verificato la coerenza.

3.4.2 Data Lifecycle

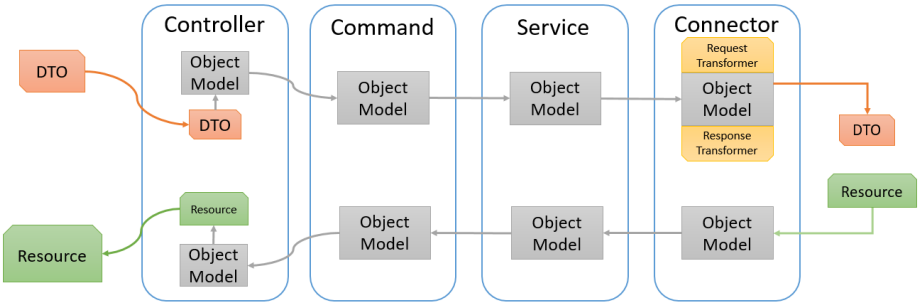

Figura 3.12: Data Lifecycle [10]

La gestione del flusso di dati tra i vari livelli dell'applicazione è stata organizzata come possiamo vedere in figura 3.12. In questo modo riusciamo a mantenere la modularità del codice e la separazione delle responsabilità senza perdere in scalabilità. Questa organizzazione garantisce lo sviluppo rispettando alcuni dei pattern architetturali dei microservizi ben noti, come:

- **Separation of Concerns (SoC)**: ogni livello ha una responsabilità chiara, riducendo l'accoppiamento tra i componenti.
- **Layered Architecture**: la suddivisione in livelli segue l'architettura stratificata tipica delle applicazioni enterprise.
- **DTO Pattern**: utilizzo di DTO per trasferire dati tra livelli senza esporre direttamente il modello di dominio.

Descrizione

L'immagine mostra il percorso che i dati compiono attraverso i vari livelli dell'applicazione, trasformandosi e adattandosi a seconda delle esigenze di ogni componente. L'architettura è suddivisa in quattro livelli principali:

- **Controller**
  - Riceve dati esterni sotto forma di DTO (Data Transfer Object) e li trasforma in un Object Model per l'elaborazione interna.
  - Restituisce le risorse ai client dopo averle convertite nel formato appropriato.
  - **Responsabilità**: gestione delle richieste HTTP, validazione dei dati e interazione con il livello di servizio.

```
1  public class StateMachinePolicyController{
2      @Autowired
3      private CreatePolicySMFactory
           createPolicySmFactory;
4
5      @PostMapping(path = "")
6      public ResponseEntity<StateMachineResource>
           createStateMachine(@RequestBody
           CreateStateMachineDto createStateMachineDTO)
           throws Exception{
7
8          PolicyStateMachineModel
               policyStateMachineModel =
               createPolicySmFactory.create(
               createStateMachineDTO);
9          PolicyStateMachineCommand
               policyStateMachineCommand = beanFactory.
               getBean(PolicyStateMachineCommand.class,
               policyStateMachineModel);
```

```
10              PolicyStateMachineOutput
                    policyStateMachineOutput =
                    createPolicyStateMachineCommand.execute();
11              CreateStateMachineAssembler
                    createStateMachineAssembler = new
                    CreateStateMachineAssembler();
12              PolicyStateMachineResource
                    policyStateMachineResource =
                    createStateMachineAssembler.toModel(
                    policyStateMachineOutput);
13
14              return new ResponseEntity<>(
                    policyStateMachineResource, HttpStatus.OK)
                    ;
15          }
16      }
```

Listing 3.12: State Machine Controller

- **Command**
  - Interagisce con il modello di dominio (Object Model) per eseguire operazioni specifiche.
  - Può essere considerato il livello applicativo che coordina le operazioni e applica la logica di business.

```
1   public class CreatePolizzeStateMachineCommand{
2
3       private PolicyStateMachineModel
            policyStateMachineModel;
4
5       @Autowired
6       private StateMachineService stateMachineService;
7
8       public PolicyStateMachineModel(
            PolicyStateMachineModel
            policyStateMachineModel) {
9           this.policyStateMachineModel =
                policyStateMachineModel;
10      }
11
12      @Override
13      protected PolicyStateMachineOutput doExecute()
            throws Exception {
14          logger.info("Create State Machine");
15          PolicyStateMachineOutput
                policyStateMachineOutput =
                stateMachineService.createStateMachine(
                policyStateMachineModel);
```

```
16
17              //spostiamo la State Machine allo stato "
                    INIZIALIZZATA"
18              MoveStateMachineModel moveStateMachineModel =
                     new MoveStateMachineModel();
19              MoveStateMachineOutput moveStateMachineOutput
                     = stateMachineService.moveStateMachine(
                    policyStateMachineOutput.getStateMachineId
                    (), "initialize", moveStateMachineModel);
20
21              return policyStateMachineOutput;
22          }
23      }
```

Listing 3.13: State Machine Command

- **Service**
  - Contiene la logica di business dell'applicazione.
  - Esegue operazioni sui dati in base alle richieste provenienti dai livelli superiori.
  - Comunica con il livello Connector per interagire con altri sistemi o database.

```
1       public class StateMachineServiceImpl{
2           @Autowired
3           private MoveStateMachineConnector
                moveStateMachineConnector;
4           @Autowired
5           private MoveStateMachineRequestTransformer
                moveStateMachineRequestTransformer;
6           @Autowired
7           private MoveStateMachineResponseTransformer
                moveStateMachineResponseTransformer;
8
9           @Override
10          public MoveStateMachineOutput moveStateMachine(
                String stateMachineId, String event,
                MoveStateMachineModel moveStateMachineModel) {
11              MoveStateMachineOutput moveStateMachineOutput
                     = moveStateMachineConnector.call(
                    moveStateMachineModel,
                    moveStateMachineRequestTransformer,
                    moveStateMachineResponseTransformer,
                    stateMachineId, event);
12              return moveStateMachineOutput;
13          }
```

```
14      }
```

Listing 3.14: State Machine Service Implementation

- **Connector**
  - Si occupa della comunicazione con sistemi esterni o database.
  - Implementa Request Transformer e Response Transformer per convertire i dati in entrata e in uscita nel formato corretto.
  - Restituisce dati sotto forma di DTO o Resource, a seconda delle necessità.

3.4.3 Connettori sincroni

REST e SOAP sono gli stili architetturali più usati nei paradigmi di comunicazione sincrona. Come abbiamo visto precedentemente, le API REST sono preferibili per la loro leggerezza e l'interoperabilità tra sistemi di natura diversa. Invece le chiamate SOAP sono utilizzate in campi in cui la sicurezza della risorsa scambiata funge un ruolo centrale. Nel nostro caso utilizziamo per la maggior parte dei servizi `REST connector` ma anche alcuni `SOAP connector` per sistemi di salvataggio documentale. I connettori sono il punto del codice nel quale prepariamo la chiamata ad un altro microservizio che può essere sia interno al nostro progetto, che un servizio offerto da un'altra società, ma di cui ne abbiamo bisogno per completare la nostra richiesta. Per utilizzare il `connector`, utilizziamo un `Request Transformer` che riceve la risorsa che dobbiamo mandare al microservizio e la incapsula all'interno di una request del formato del microservizio stesso. In questi casi è molto importante avere una documentazione precisa e completa sul servizio che dobbiamo chiamare. Per agevolare queste operazioni vengono usati strumenti come Swagger. Uno Swagger offre una serie di funzionalità per documentare e testare API RESTful. Permette di testare le API che vogliamo chiamare, senza dover accedere al servizio esterno, consultando una documentazione chiara e leggibile . Offre anche tool che permettono di generarla automaticamente dal nostro codice senza che il programmatore se ne occupi esplicitamente. Così, grazie a questi strumenti, riusciamo a comunicare anche con servizi di cui non conosciamo l'implementazione. Una volta inviata la richiesta al servizio dobbiamo rimanere in attesa di una risposta. Il `Response Transformer` si occuperà di controllare il formato della risposta e di formattarla nell'oggetto di cui abbiamo bisogno. Di solito la response contiene, oltre la richiesta di cui avevamo bisogno, il codice di stato oppure un messaggio di errore in caso di fallimento.

Di seguito possiamo vedere il codice inerente ai connettori sincroni che abbiamo utilizzato nel nostro progetto. Il primo esempio (3.15) mostra la `moveStateMachine` che avevamo analizzato in precedenza. Creiamo un `RestConnectorRequest` che sarà formattato in un formato HTTP per essere passato al servizio di interesse. Nel method inseriamo il tipo di chiamata, nel payload la risorsa, nella URI l'indirizzo univoco corrispondente al servizio e riempiremo eventuali headers e params. Il connector una volta mandata la request appena creata aspetterà, in modo bloccante, una response dal microservizio chiamato. Per interpretare ciò che riceveremo c'è bisogno di un `RespConnectorResponse` che permetta di formattare nel giusto modo la risposta. In questo caso estraiamo la response dall'oggetto ricevuto per settare il nostro output, formato dallo stato precedente della State Machine e dallo stato attuale in cui ci siamo voluti spostare.

```
1  public class RestConnectorRequest<INPUT> {
2      private HttpHeaders httpHeaders;
3      private INPUT payload;
4      private Map<String, String> params = new HashMap();
5      private HttpMethod method;
6      private String url;
7  }
8  public class MoveStateMachineRequestTransformer{
9      @Override
10     public RestConnectorRequest<
           MoveStateMachinePayloadRequest>
11         transform(MoveStateMachineModel moveStateMachineModel
               , Object... args) {
12
13         RestConnectorRequest<MoveStateMachinePayloadRequest>
               restConnectorRequest = new RestConnectorRequest
               <>();
14         restConnectorRequest.setDynamicMethod(HttpMethod.POST
               );
15         restConnectorRequest.setRequest(moveStateMachineModel
               );
16         restConnectorRequest.addParams("stateMachineId", args
               [0].toString());
17         restConnectorRequest.addParams("event", args[1].
               toString());
18
19         return restConnectorRequest;
20     }
21 }
22
23 public class MoveStateMachineResponseTransformer{
24     @Override
```

```
25      public MoveStateMachineOutput transform(
            RestConnectorResponse<MoveStateMachineResource>
            response) {
26          MoveStateMachineOutput moveStateMachineOutput = new
                MoveStateMachineOutput();
27          if(response != null && response.getResponse() != null
                && response.getResponse().getBody() != null) {
28          moveStateMachineOutput.setPreviousState(response.
                getResponse().getBody().getPreviousState());
29          moveStateMachineOutput.setCurrentState(response.
                getResponse().getBody().getCurrentState());
30          }
31
32          return moveStateMachineOutput;
33  }
```

Listing 3.15: State Machine Request and Response

Negli esempi successivi(3.16) possiamo vedere la classe dei SOAP Connector. Nella richiesta oltre al payload inseriamo l'URI, gli HTTP headers e in più, questa volta, i SOAP headers. Anche la funzione `transform` è molto simile alla precedente in quanto va a riempire l'oggetto request con i dati che riceve. La differenza fondamentale si trova nella formattazione della request, in questo caso verrà formattata in formato XML e mandata al microservizio SOAP. Rimarremo in attesa di una response che verrà manipolata dal `transform` del `ResponseTranformer`.

```
1   public class SoapConnectorRequest<INPUT>{
2       private INPUT payload;
3       private SoapHeader soapHeader;
4       private String url;
5       private List<Header> httpHeaders;
6   }
7   @Override
8   public SoapConnectorRequest<InsertDoc> transform(InsertDoc
        insertDoc, Object... args) {
9       SoapConnectorRequest<InsertDoc> request = new
            SoapConnectorRequest<>();
10      request.setSoapHeader(null);
11      request.setPayload(insertDoc);
12
13      return request;
14  }
15  @Override
16  public InsertDocOut transform(SoapConnectorResponse<
        InsertDocResponse> response) {
17      if(response == null || response.getResponse() == null ||
            response.getResponse().getInsertDocResponse() == null)
            {
```

```
18          throw new RuntimeException("NO_RESPONSE_RECEIVED", "
                No response received - Insert document soap
                connector", HttpStatus.INTERNAL_SERVER_ERROR);
19      }
20
21      return response.getResponse().getInsertResponsePart();
22  }
```

Listing 3.16: SOAP connector

3.4.4 Connettori asincroni

Non sempre la comunicazione sincrona, sebbene sia semplice e veloce, riesce a soddisfare le esigenze del programmatore. In casi in cui dobbiamo mantenere una comunicazione real time e gestire dipendenze tra più microservizi diversi è più utile utilizzare protocolli asincroni basati su eventi. In questo caso ricorriamo ai connettori Kafka che, come abbiamo visto in precedenza, permettono a più microservizi di scrivere su uno stesso topic garantendo la persistenza dei dati. Ogni servizio potrà leggere o scrivere dal topic iscrivendosi ad esso. Questi connettori sono stati utilizzati soprattutto nelle coreografie funzionali di firma e generazione documentale. Per utilizzare Kafka bisogna innanzitutto definire nell' `application.yml` i connettori che andremo ad utilizzare, i consumer e i producer. Qui di seguito possiamo vedere un esempio sull'avvio firma (3.17). Oltre a definire il nome del connettore e il suo ruolo dobbiamo specificare l'endpoint, nel quale andranno definiti: il topic del quale fanno parte, il groupId e il broker principale, in questo caso Kafka. Naturalmente entrambi i servizi dovranno iscriversi allo stesso topic per comunicare tra loro.

```
1  {
2  integration:
3      connectors:
4          kafka:
5              consumers:
6                  AvvioFirmaConsumer:
7                      enabled: true
8                      endpoint:
9                          topic: AVVIO_FIRMA_TOPIC
10                         groupId: FIRMA_POLIZZA
11                         brokers: KAFKA_POLIZZA_BROKERS
12                         consumersCount: 1
13                     processorBeanRef: AvvioFirmaProcessor
14                     messageFormat: JSON
15             producers:
16                 AvvioFirmaProducer:
17                     processorBeanRef: AvvioFirmaProcessor
```

```
                    enabled: true
                    endpoint:
                        topic: AVVIO_FIRMA_TOPIC
                        brokers: KAFKA_POLIZZA_BROKERS
                    errorHandling:
                        enabled: true
                        redeliveryDelay: 500
                        maxRetry: 10
}
```

Listing 3.17: Kafka application.yml

Una volta definiti producer e consumer nell'`application.yml` possiamo creare un Event (3.18 da poter inviare sull'Event Bus. Creiamo l'oggetto `EventRequest` che sarà inviato dal connettore. Gli attributi più importanti sono: il message che conterrà il payload del messaggio che dobbiamo inviare, gli headers e il topic su cui inviare il messaggio.

```
public  class EventRequest<DTO>{
    private String key;
    private Headers headers;
    private String topic;
    protected DTO payload;
}
@Override
public EventRequest<Void> transform(NotificationEventInput
   eventInput, Object... args) {
    EventRequest<Void> event = new EventRequest<>();
    List<Header> headers = new ArrayList<>();
    Headers h = new RecordHeaders(headers);
    String message = eventInput.message.toString();
    event.setPayload(message.getBytes(StandardCharsets.UTF_8)
       );
    event.setTopic(eventInput.topic);
    event.setHeaders(h);

    return event;
}
```

Listing 3.18: Kafka connector

3.4.5 Coreografie Funzionali

Nel campo dei microservizi, le coreografie funzionali offrono un approccio decentralizzato alla comunicazione tra i servizi. Questo metodo permette di scambiare informazioni tra di essi attraverso meccanismi di comunicazione asincrona. Si oppone alle architetture centralizzate in cui è presente un orchestratore che gestisce la comunicazione in un ambiente centralizzato. Anche

in questo caso, quando si tratta di scegliere quale delle due strade seguire bisogna analizzare le necessità del progetto e decidere a cosa si può rinunciare. Analizzando la ricerca[12], sui casi di utilizzo dei metodi, abbiamo scelto l'uso di coreografie per gestire gli eventi che includono più microservizi e non possono comunicare tramite chiamate sincrone (come la generazione dei documenti e la gestione della firma).

Generazione documentale

Nei sistemi basati su microservizi, l'uso di chiamate sincrone e bloccanti può portare a problemi di scalabilità, aumentando il rischio di timeout ed errori. Per evitare questi problemi, un approccio più efficace è quello basato su eventi, utilizzando un event bus per orchestrare la comunicazione tra i servizi. Nel caso della generazione documentale, il processo si articola nei seguenti passi:

- Il sistema invia un evento sul bus dei messaggi con il comando **GEN DOCUMENTS**, notificando la necessità di generare i documenti per una polizza.
- Il microservizio responsabile della generazione intercetta questo evento e avvia il processo di creazione. Una volta completato, salva i documenti nel proprio sistema di archiviazione.
- Una volta pronti, il servizio di generazione emette un nuovo evento, **GEN DOCUMENTS OK**, per informare gli altri microservizi che i documenti sono disponibili.
- Il servizio documentale intercetta questo evento e si occupa di prelevare i documenti generati, associandoli alla polizza corrispondente.
- Se l'operazione ha successo, viene generato l'evento **OFFERTA DISPONIBILE**, che segnala il completamento della fase documentale. Prima di emettere questa notifica, vengono eliminati eventuali file temporanei dal servizio di generazione.
- Infine, il Back End intercetta l'evento **OFFERTA DISPONIBILE** e aggiorna lo stato della polizza, avanzando la *State Machine* verso lo stato **DOCUMENTAZIONE DISPONIBILE**.

L'uso di un'architettura basata su eventi per la gestione della generazione documentale porta diversi vantaggi:

- **Scalabilità**: i microservizi comunicano in modo asincrono, evitando blocchi nel flusso di esecuzione.
- **Gestione efficiente degli errori**: il sistema può gestire eventuali ritardi o errori, riprovando l'elaborazione degli eventi senza interrompere l'intero processo.
- **Indipendenza tra servizi**: ogni servizio è indipendente dagli altri e non deve conoscere i dettagli interni degli altri microservizi.

Questo approccio consente quindi di gestire in modo efficiente la generazione e l'associazione dei documenti alla polizza, migliorando la robustezza e la manutenibilità del sistema.

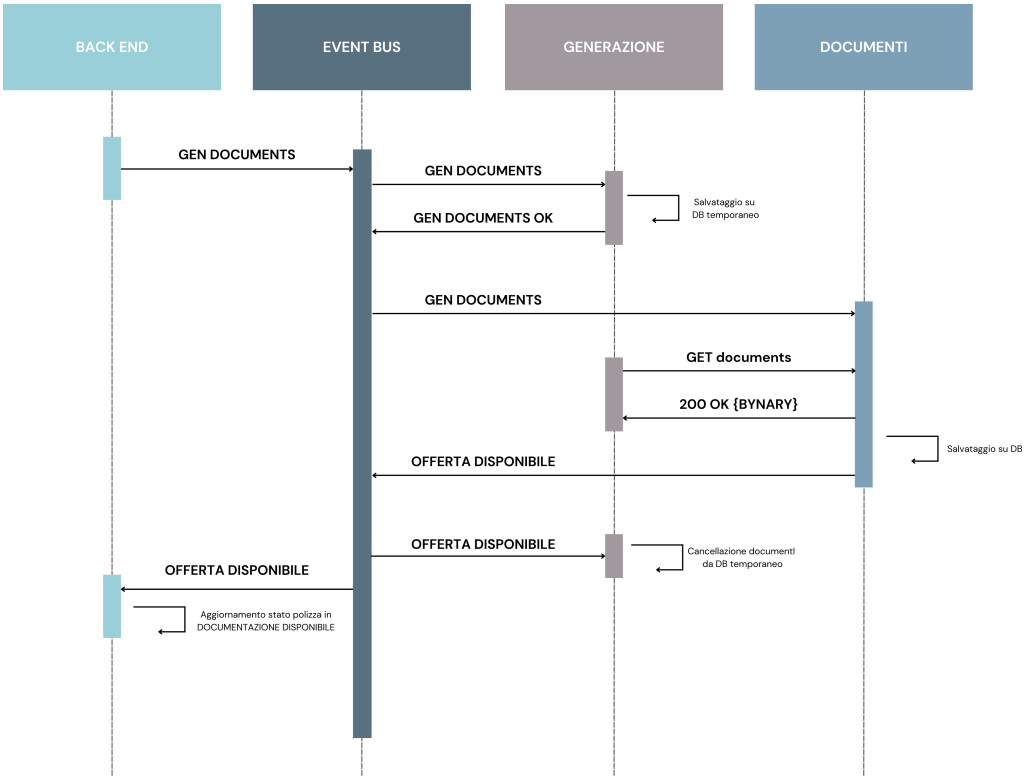

Figura 3.13: Generazione Documentale

Avvio firma

Una volta che tutti i documenti sono stati consultati, il sistema deve avviare il processo di firma. Anche in questo caso, per garantire scalabilità e

indipendenza tra i microservizi, viene utilizzato un event bus per gestire la comunicazione tra i servizi coinvolti.

Il flusso del processo di avvio firma è il seguente:

- Il nostro sistema avvia la procedura inviando un evento **AVVIO FIRMA** sul bus dei messaggi.
- Il primo microservizio che intercetta questo evento è il servizio che si occupa dei documenti, il quale recupera tutti i documenti relativi alla polizza da firmare.
- Una volta ottenuti i documenti, il servizio emette l'evento **DOCUMENTI PRONTI OK**, segnalando che i file necessari sono disponibili.
- Il servizio di firma intercetta questo evento e avvia il processo di firma digitale, salvando e aggiornando il sistema con tutti i documenti che andranno firmati.
- Una volta che tutto è pronto viene inviato l'evento **OFFERTA DISPONIBILE**.
- Il Back End intercetta l'evento e aggiorna lo stato della polizza, avanzando la *State Machine* allo stato **IN FIRMA**.

L'uso dell'event bus in questo scenario offre diversi vantaggi:

- **Automazione**: il processo di avvio firma viene eseguito automaticamente al verificarsi delle condizioni necessarie, senza intervento manuale.
- **Gestione asincrona**: ogni servizio elabora i propri compiti senza bloccare il flusso o intervenire su altri processi, così riusciamo ad evitare errori o rischi di timeout.
- **Affidabilità**: la gestione degli eventi permette di tenere sotto controllo lo stato del processo e di intervenire su eventuali problemi.

Grazie a questa architettura, il sistema garantisce una gestione efficiente e robusta della firma dei documenti, assicurando la corretta progressione della polizza nel suo ciclo di vita.

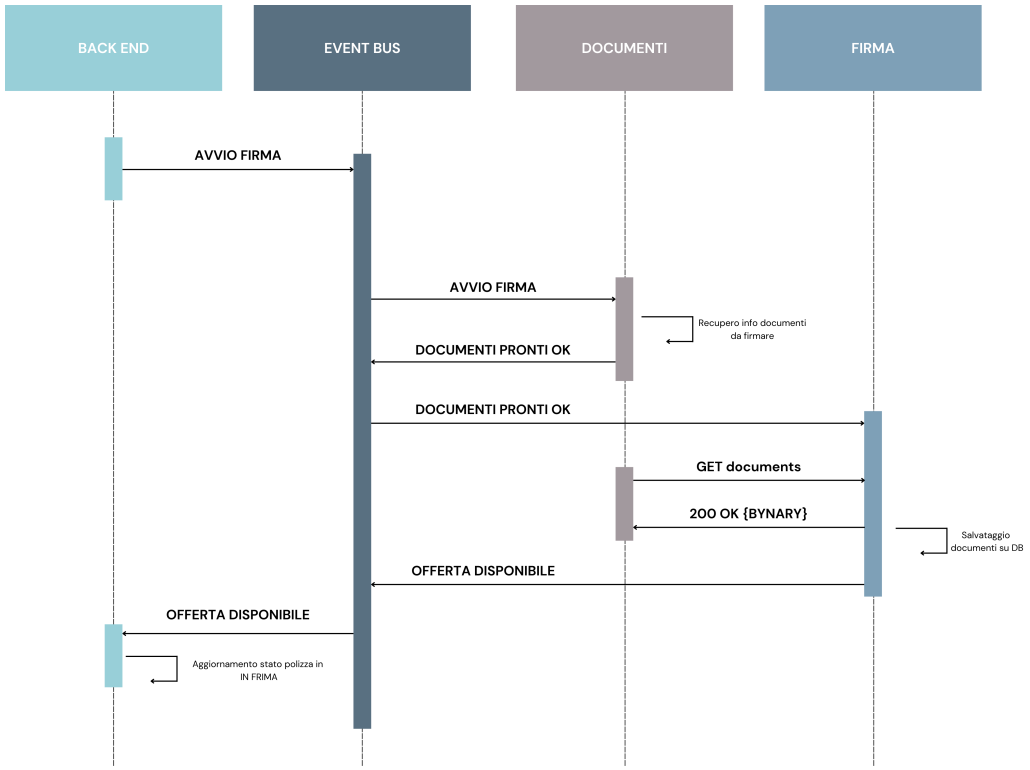

Figura 3.14: Avvio Firma

Capitolo 4

Testing

La fase di test è molto importante nel ciclo di vita di sviluppo del software, indispensabile per assicurare la qualità del codice, l'affidabilità e la manutenibilità del prodotto finale. In un contesto in cui le applicazioni sono strutturate in microservizi e ambienti separati, come in Spring Boot, adottare una giusta strategia di testing è cruciale per agire in modo mirato e veloce sugli errori e problemi che si verificano. Studi accademici, come quello di Myers [13], evidenziano l'importanza di integrare il testing sin dalle prime fasi di progettazione, riducendo i costi di manutenzione e il debito tecnico che potrebbe aumentare con il passare del tempo. Avere una buona copertura di test non solo è fondamentale per garantire la qualità del software ma anche per abilitare il caricamento del codice sul flusso di integrazione e deployment continuo (**CI/CD**). Questi sistemi richiedono infatti un'alta percentuale di test riusciti affinché possano distribuire in sicurezza le nuove versioni del software, in quanto correggere e debuggare durante i test automatici è molto più facile ed meno costoso rispetto ad agire una volta che il codice è stato rilasciato in produzione.
Ci sono vari tipi di test, ognuno dei quali mira ad un aspetto diverso del software: **Unit Tests**, **Integration Tests** e **UI Tests**. Il primo si concentra sul funzionamento della singola unità del codice, il secondo verifica l'interazione tra diversi componenti ricreando l'ambiente di produzione che stiamo creando mentre l'ultimo può simulare il comportamento del software finale andando a testare contemporaneamente Ui e Back End. Come notiamo nella figura (4.1), ogni tipologia ha i suoi costi e le sue difficoltà. Partendo dal basso capiamo che gli unit test sono i più semplici poiché vanno a testare il singolo componente. Proprio per questo dovranno essere maggiori in numero risparmiando in tempo e costo. Risalendo la piramide vediamo come la difficoltà dei test aumenta, dovendo integrare e testare contemporaneamente diversi componenti. Queste metodologie, supportate da strumenti

come JUnit e Mockito, sono integrate in un contesto di sviluppo moderno che permette l'automazione e la continuità nella distribuzione del codice.

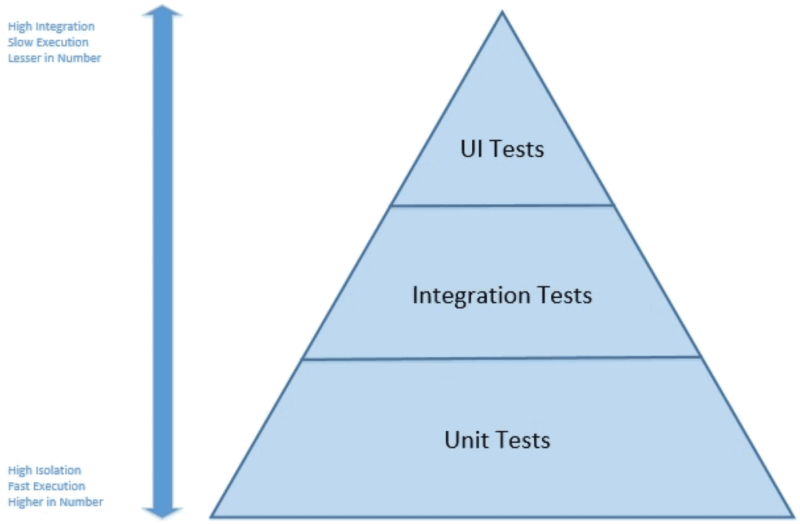

Figura 4.1: Pyramid of testing [4]

4.1 Unit Testing

Gli Unit testing hanno l'obiettivo di validare il comportamento di singole unità di codice, come metodi o classi, isolandole dalle dipendenze esterne. Utilizzando framework come **JUnit** e **Mockito**, è possibile simulare il comportamento delle dipendenze, come ad esempio repository e servizi esterni, permettendo di controllare esclusivamente la logica implementata nello specifico pezzo di codice. L'impiego dei mock è fondamentale perché consente di ottenere test che eliminano i collegamenti con servizi esterni. Come evidenziato dallo studio sui mock [20], l'utilizzo di mock contribuisce a ridurre il debito tecnico e a migliorare la copertura dei test. Per ogni componente del Data Lifecycle abbiamo scritto test appositi che, rispettando le precedenti linee guida, hanno aiutato a sviluppare in maniera più corretta e lineare il software. Ogni test è molto utile anche in fase di scrittura di un metodo in quanto permette di capire sin da subito se sono stati commessi errori o ci sono potenziali vulnerabilità nel codice.
Abbiamo implementato vari test che coprono tutte le diverse componenti del sistema inclusi i Model, DTO, Factory, Assembler, Command e Connector. Ogni classe di test verifica il corretto funzionamento delle rispettive componenti, assicurandosi che vengano gestiti correttamente gli oggetti e le

interazioni tra i vari livelli dell'applicazione. Mostreremo in seguito alcuni dei test implementati.

Unit Test Model

Iniziamo con il test della classe `PolicyStateMachineModel`. Questo test verifica la corretta inizializzazione degli oggetti `MoveStateMachineOutput` e assicura che i metodi di accesso restituiscano i valori corretti per i vari campi, come `previousState` e `currentState`.

```
1
2  public class PolicyStateMachineModelTest {
3      @Test
4      public void moveStateMachineOutputTest(){
5          logger.info("Start moveStateMachineOutputTest");
6
7          MoveStateMachineOutput moveStateMachineOutput = new
               MoveStateMachineOutput();
8          moveStateMachineOutput.setPreviousState("previous");
9          moveStateMachineOutput.setCurrentState("current");
10
11         assertNotNull(moveStateMachineOutput);
12         assertEquals("previous", moveStateMachineOutput.
               getPreviousState());
13         assertEquals("current", moveStateMachineOutput.
               getCurrentState());
14     }
15 }
```

Listing 4.1: Unit Test Model

Unit Test DTO

Passiamo ora al test della classe `PolicyStateMachineDto`. Questo test verifica che i vari campi del DTO, come `channel`, `offerType`, `signType`, e gli oggetti associati come UserDto, vengano correttamente inizializzati e assegnati.

```
1
2  public class PolicyStateMachineDtoTest {
3      @Test
4      public void createPolizzeStateMachineDtoTest(){
5          logger.info("Start createPolizzeStateMachineDtoTest")
               ;
6
7          CreatePolizzeStateMachineDto
               createPolizzeStateMachineDto = new
               CreatePolizzeStateMachineDto();
```

```
8           createPolizzeStateMachineDto.setChannel("web");
9           createPolizzeStateMachineDto.setOfferType("
                multicanale");
10          createPolizzeStateMachineDto.setSignType("remota");
11          UserDto userDto = new UserDto();
12          createPolizzeStateMachineDto.setUser(userDto);
13          Map<String, String> additionalInfo = new HashMap<>();
14          additionalInfo.put("key", "value");
15          createPolizzeStateMachineDto.setAdditionalInfo(
                additionalInfo);
16
17          assertNotNull(createPolizzeStateMachineDto);
18          assertEquals("web", createPolizzeStateMachineDto.
                getChannel());
19          assertEquals("multicanale",
                createPolizzeStateMachineDto.getOfferType());
20          assertEquals("remota", createPolizzeStateMachineDto.
                getSignType());
21          assertEquals(userDto, createPolizzeStateMachineDto.
                getUser());
22          assertEquals(additionalInfo,
                createPolizzeStateMachineDto.getAdditionalInfo());
23
24          createPolizzeStateMachineDto = new
                CreatePolizzeStateMachineDto(
25                  "web",
26                  "multicanale",
27                  "remota",
28                  userDto,
29                  additionalInfo
30          );
31
32          assertNotNull(createPolizzeStateMachineDto);
33      }
34  }
```

Listing 4.2: Unit Test Dto

Unit Test Factory

Il test della classe `PolicyStateMachineFactory` verifica che la factory riesca a creare correttamente un'istanza di `PolicyStateMachineModel` partendo da un DTO. Controlla anche il comportamento quando alcuni campi, come `contextDTO`, vengono impostati a null.

```
1
2   public class PolicyStateMachineFactoryTest {
3
```

```
4       @Test
5       public void createPolizzeStateMachineFactoryTest() throws
            Exception{
6           logger.info("Start
                createPolizzeStateMachineFactoryTest");
7
8           CreatePolizzeStateMachineDto
                createPolizzeStateMachineDto = new
                CreatePolizzeStateMachineDto();
9           createPolizzeStateMachineDto.setChannel("web");
10          createPolizzeStateMachineDto.setOfferType("
                multicanale");
11          createPolizzeStateMachineDto.setSignType("remota");
12          UserDto userDto = new UserDto();
13          userDto.setFiscalCode("fiscalCode");
14          userDto.setUserId("userId");
15          createPolizzeStateMachineDto.setUser(userDto);
16          createPolizzeStateMachineDto.setAdditionalInfo(new
                HashMap<>());
17
18          PolicyStateMachineModel policyStateMachineModel =
                createPolizzeStateMachineFactory.create(
                createPolizzeStateMachineDto);
19          Assert.assertNotNull(policyStateMachineModel);
20      }
21  }
```

Listing 4.3: Unit Test Factory

4.1.1 Unit Test Assembler

Il test dell'assembler si assicura che l'oggetto **PolicyStateMachineOutput** venga correttamente trasformato in un oggetto **StateMachineResource**, utilizzando la logica di conversione definita nell'assembler.

```
1
2   public class PolicyStateMachineAssemblerTest {
3       @Test
4       public void createPolizzeStateMachineAssemblerTest() {
5           logger.info("Start
                createPolizzeStateMachineAssemblerTest");
6
7           StateMachineResource policyStateMachineOutput = new
                PolicyStateMachineOutput();
8           policyStateMachineOutput.setStateMachineId("123456");
9           CreatePolizzeStateMachineAssembler
                createPolizzeStateMachineAssembler = new
                CreatePolizzeStateMachineAssembler();
```

```
10
11          StateMachineResource stateMachineResource =
                createPolizzeStateMachineAssembler.toModel(
                policyStateMachineOutput);
12          Assert.assertNotNull(stateMachineResource);
13      }
14  }
```

Listing 4.4: Unit Test Assembler

4.1.2 Unit Test Command

Nel test del `PolicyStateMachineCommand`, verifichiamo che il comando `CreatePolizzeStateMachineCommand` esegua correttamente l'operazione, invocando i metodi appropriati sul `stateMachineService` e restituendo il risultato atteso. In questo contesto riusciamo a vedere l'importanza dell'uso dei Mock. Infatti per isolare il test e verificare solo il comportamento del command, senza dipendenze dai servizi esterni, mockiamo le risposte dei metodi chiamati. In questa classe il test coinvolge due chiamate: una al servizio `createStateMachine` e una al servizio `moveStateMachine`. Grazie ai Mock, possiamo simulare il comportamento di questi servizi e fornirgli un valore di ritorno predefinito con `doReturn`. Così, quando i metodi vengono invocati, non chiamano i veri servizi, ma restituiscono gli oggetti che abbiamo definito, permettendoci di testare esclusivamente il comportamento del `PolicyStateMachineCommand`.

```
1
2   public class PolicyStateMachineCommandTest {
3       @Test
4       public void createPolizzeStateMachineCommandTest() throws
            Exception{
5           logger.info("Start
                createPolizzeStateMachineCommandTest");
6
7           PolicyStateMachineModel policyStateMachineModel = new
                PolicyStateMachineModel();
8           createPolizzeStateMachineCommand = new
                CreatePolizzeStateMachineCommand(
                policyStateMachineModel);
9           MockitoAnnotations.openMocks(this);
10
11          MoveStateMachineOutput moveStateMachineOutput= new
                MoveStateMachineOutput("prev", "curr");
12          Mockito.doReturn(new PolicyStateMachineOutput()).when
                (stateMachineService).createStateMachine(Mockito.
                any());
```

```
13          Mockito.doReturn(moveStateMachineOutput).when(
                stateMachineService).moveStateMachine(Mockito.any
                (), Mockito.any(), Mockito.any());
14          Assert.assertNotNull(createPolizzeStateMachineCommand
                .execute());
15      }
16  }
```

Listing 4.5: Unit Test Command

Per testare il connector vengono testati separatamente prima la request e successivamente la response. La classe di test **PolicyStateMachineRequestTransformer** verifica che un oggetto **PolicyStateMachineModel** venga correttamente trasformato in una richiesta di tipo **RestConnectorRequest**. La classe **PolicyStateMachineResponseTransformer** verifica il corretto passaggio da **RestConnectorResponse** a **PolicyStateMachineOutput**. In questo caso possiamo vedere anche la gestione delle eccezioni, nel caso in cui riceviamo una response null otterremo un errore del tipo: `"No stateMachineId or policyId received"`.

```
1
2   public class  PolicyStateMachineRequestTransformerTest {
3       @Test
4       public void createStateMachineRequestTransformerTest(){
5           logger.info("Start
                createStateMachineRequestTransformerTest");
6
7           PolicyStateMachineModel policyStateMachineModel = new
                PolicyStateMachineModel();
8           User user = new User("userId", "codFiscale");
9           policyStateMachineModel.setUserModel(user);
10          policyStateMachineModel.setChannel("web");
11          policyStateMachineModel.setSignType("remota");
12          policyStateMachineModel.setOfferType("multicanale");
13          Map<String, String> addInfo = new HashMap<>();
14          addInfo.put("add", "info");
15          policyStateMachineModel.setAdditionalInfo(addInfo);
16
17          RestConnectorRequest<CreateStateMachinePayloadRequest
                <>> request = createStateMachineRequestTransformer
                .transform(policyStateMachineModel);
18          Assert.assertNotNull(request);
19      }
20  }
21
22  public class  PolicyStateMachineRequestTransformerTest {
23  @Test
24      public void createStateMachineResponseTransformerTest(){
```

```
25          logger.info("Start
               createStateMachineResponseTransformerTest");
26
27          PolicyStateMachineResource policyStateMachineResource
                = new PolicyStateMachineResource();
28          policyStateMachineResource.setStateMachineId("
               stateMachineId");
29          policyStateMachineResource.setPolicyId("policyId");
30
31          ResponseEntity<PolicyStateMachineResource>
               responseEntity = new ResponseEntity<>(
               PolicyStateMachineResource, HttpStatus.OK);
32          RestConnectorResponse<PolicyStateMachineResource>
               restConnectorResponse = new RestConnectorResponse
               <>();
33          restConnectorResponse.setResponse(responseEntity);
34
35          CreateStateMachineResponseTransformer
               createStateMachineResponseTransformer = new
               CreateStateMachineResponseTransformer();
36          PolicyStateMachineOutput response =
               createStateMachineResponseTransformer.transform(
               restConnectorResponse);
37
38          Assert.assertNotNull(response);
39
40          Assert.assertThrows("No stateMachineId or policyId
               received", CreateStateMachineException.class, ()
               ->
41          {
42              restConnectorResponse.setResponse(null);
43              createStateMachineResponseTransformer.transform(
                   restConnectorResponse);
44          });
45      }
46  }
```

Listing 4.6: Unit Test Connector

4.2 Integration Testing in Spring Boot

L'integration testing si focalizza sulla verifica dell'interazione tra i vari componenti dell'applicazione. In un ambiente Spring Boot, questo tipo di test è fondamentale per garantire che controller, servizi e repository collaborino correttamente. Simuleremo un ambiente reale, in cui una richiesta HTTP passa attraverso l'intero stack, dal controller fino ai servizi che utilizza. Di

seguito, un esempio che dimostra come testare un controller relativo alla State Machine:

```
1
2  public class StateMachinePolicyControllerTest {
3      @Test
4      public void createPolizzeStateMachineTest() throws
           Exception{
5          logger.info("Start createPolizzeStateMachineTest");
6
7          CreatePolizzeStateMachineDto
               createPolizzeStateMachineDto = new
               CreatePolizzeStateMachineDto();
8
9          createPolizzeStateMachineDto.setChannel(Canale.ABC);
10         createPolizzeStateMachineDto.setOfferType(OfferType.
               MULTICANALE);
11         createPolizzeStateMachineDto.setSignType(SignType.FEA
               );
12         createPolizzeStateMachineDto.setCustomer(new
               CustomerDto());
13         createPolizzeStateMachineDto.setUser(new UserDto());
14         createPolizzeStateMachineDto.setContextDTO(new
               ContextDto());
15         createPolizzeStateMachineDto.setAdditionalInfo(new
               HashMap<>());
16
17         String URI = "/state-machine/polizze";
18
19         MvcResult mvcResult = mockMvc.perform(
20                     MockMvcRequestBuilders.post(URI)
21                         .contentType(MediaType.
                               APPLICATION_JSON_VALUE)
22                         .content(toString(
                               createPolizzeStateMachineDto))
23                         .accept(MediaType.
                               APPLICATION_JSON_VALUE)
24                         .accept(MediaTypes.HAL_JSON))
25                 .andExpect(MockMvcResultMatchers.status().
                       isOk())
26                 .andReturn();
27         CreatePolizzeStateMachineResource response = toObject
               (mvcResult, CreatePolizzeStateMachineResource.
               class);
28         Assertions.assertNotNull(response);
29     }
30
31 }
```

Listing 4.7: Integration Test Controller

Questo test sul controller della State Machine verifica che la chiamata all'endpoint **/state-machine/polizze** restituisca uno stato HTTP 200 e che il contenuto JSON contenga l'identificativo corretto, assicurando così che il flusso di dati tra i componenti funzioni come previsto.

4.3 Utilizzo di Strumenti Avanzati e DevOps

Per ottimizzare il processo di sviluppo e testing del software sono stati affiancati al nostro progetto strumenti avanzati come GitHub Copilot, che fornisce assistenza nella scrittura dei test e nella compilazione di codice boilerplate. Il maggior utilizzo è stato nella generazione di test ripetitivi su classi che definiscono strutture o implementano metodi vuoti o semplici getter e setter. Contemporaneamente l'utilizzo di una pipeline CI/CD automatizzata permette di tenere costantemente i test sopra una certa soglia di riuscita, durante il ciclo di vita dello sviluppo, garantendo un feedback immediato e continuo agli sviluppatori. Tale integrazione è fondamentale per mantenere elevata la qualità del codice e ridurre i tempi di debug e rilascio.

4.4 Implementazione della Pipeline CI/CD

La pipeline CI/CD è un elemento essenziale per garantire la continuità e l'automazione nel processo di build, test e deployment. Nel nostro progetto è stata integrata grazie al moderno software Jenkins che permette di gestire in modo facile e intuitivo il deploy del codice [14]. In dettaglio la nostra pipeline prevede diverse fasi:

- **Build:** compilazione del codice e esecuzione automatizzata di unit test e integration test, con report di copertura su SonarQube.
- **Deploy:** rilascio automatico in ambienti di test, rendendo possibile l'interazione con altri microservizi.
- **Promotion:** rilascio automatizzato in ambienti di produzione dopo aver passato con successo i vari test di ambiente.

Questa automazione consente di ottenere un feedback immediato sulle modifiche apportate, migliorando la qualità del software e riducendo significativamente i tempi di rilascio.

4.5 Monitoraggio e Troubleshooting

In ambienti complessi, risolvere i problemi richiede l'uso di strumenti e metodi sofisticati che facilitino un'analisi approfondita e interventi tempestivi. Splunk, ad esempio, viene impiegato per centralizzare il monitoraggio dei log, offrendo dashboard interattive e notifiche automatiche in caso di errori critici. E' una delle più importanti piattaforme di riferimento nel settore, permettendo di analizzare e visualizzare i dati generati dalla comunicazione tra vari servizi nelle applicazioni. Nel nostro progetto, Splunk è stato scelto per raccogliere, analizzare e correlare i log provenienti dai diversi microservizi, consentendo così un controllo dettagliato delle performance e l'individuazione rapida di eventuali criticità[1]. Splunk garantisce all'applicazione alcuni vantaggi significativi:

- **Centralizzazione dei log:** tutti i log di chiamata tra microservizi vengono inviati a Splunk, eliminando la necessità di accedere ai singoli container o server.
- **Query avanzate e dashboard personalizzate:** attraverso le SPL (Search Processing Language) di Splunk è possibile filtrare e analizzare rapidamente i log.
- **Monitoraggio in tempo reale:** gli amministratori possono settare alert per rilevare velocemente anomalie di sistema.
- **Correlazione degli eventi:** possibilità di unificare i log di più microservizi per individuare la causa radice dei problemi.

L'uso di Splunk può portare molti benefici nello sviluppo di software e ricerca errori. Grazie alla console disponibile è molto facile risalire ad errori che si sono verificati. E' possibile tenere sotto controllo l'intero sistema consultando anche eventuali accessi indesiderati e tentativi di richieste eccessive in entrata da parte di qualche sistema malevolo.

4.5.1 Troubleshooting con AI agent

Nel nostro progetto è stato integrato un AI Agent per analizzare i log estratti da Splunk e identificare eventuali errori in produzione. Questo strumento sfrutta algoritmi di Machine Learning per fornire suggerimenti e possibili soluzioni, facilitando così il lavoro dei programmatori durante il troubleshooting. Negli ultimi anni, l'uso di AI Agent si è diffuso notevolmente nelle aziende come supporto per accelerare il processo di individuazione e risoluzione degli errori. La nostra azienda intende, infatti, aumentare l'adozione

dell'Intelligenza Artificiale nella progettazione e nello sviluppo quotidiano, puntando a integrare questi strumenti come parte integrante dei processi di lavoro.

In questo contesto, abbiamo integrato un AI Agent, creato da un nostro collega, che consente di analizzare i log attraverso una semplice chiamata via Postman. L'Agent si occupa di interrogare Splunk per recuperare i log relativi a specifici pattern durante precisi intervalli temporali e restituire un'analisi strutturata. In particolare, l'output dell'Agent indica se l'operazione è andata a buon fine o se si sono verificati errori. In caso di esito negativo, l'output fornisce anche il tipo di errore e una possibile soluzione suggerita da GPT-4 di OpenAI.

Implementazione

L'AI Agent nasce da una ricerca mirata all'introduzione di nuove evoluzioni dell'intelligenza artificiale applicabili ai processi quotidiani del lavoro in ufficio. Lo script, sviluppato in Python, si divide in tre fasi principali:

1. **Elaborazione dei dati di ingresso:** Il sistema analizza il messaggio ricevuto in input per estrarre le informazioni chiavi come l'acronimo del componente, l'ambiente di esecuzione in cui cercare, il path dell'API e l'intervallo temporale su cui effettuare la ricerca nei log.

2. **Connessione a Splunk e invio della ricerca:** L'Agent si collega a Splunk e genera una query basata sui parametri ottenuti in precedenza. Ritornando i log relativi all'intervallo specificato.

3. **Analisi dei risultati:** I log raccolti vengono inoltrati a OpenAI per la decodifica. L'output finale restituito consta di una risposta strutturata che include un'analisi dettagliata dei dati e, in caso di errori, dettagli e suggerimenti per la loro risoluzione.

Un esempio di chiamata per l'analisi dei log è la seguente:

```
{
    "message": "I need to understand what happened on the
       component with acronim policy and env utes and path
       policyInfo, between 16:30 and 16:40"
}
```

Listing 4.8: Esempio chiamata

L'Agent, con il supporto di OpenAI, elabora il messaggio in ingresso per generare una rappresentazione strutturata in formato JSON. Questo JSON

contiene tutti i parametri estratti dall'input, come l'acronimo del componente, l'ambiente di esecuzione, il percorso dell'API e l'intervallo temporale specificato dall'utente.

Una volta generato, il JSON viene utilizzato per costruire una query completa che include tutte le informazioni necessarie per interrogare Splunk in modo efficiente. La query viene arricchita con i dettagli relativi alla sessione e ai cookie, garantendo così una richiesta corretta e contestualizzata.

Successivamente, la richiesta viene inviata a Splunk, che restituisce i log pertinenti all'intervallo di tempo selezionato. I log recuperati vengono poi inoltrati a ChatGPT per l'analisi semantica e la decodifica delle informazioni contenute. Il sistema è in grado di identificare pattern ricorrenti, errori di sistema e possibili anomalie nei dati analizzati.

L'output finale generato dall'Agent può variare in base al contenuto dei log recuperati. Se vengono rilevati errori, come ad esempio un errore HTTP 500, la risposta includerà un'analisi dettagliata del problema, evidenziando la causa dell'anomalia e suggerendo possibili azioni correttive. Se invece non emergono errori evidenti, il sistema può comunque fornire informazioni utili sulla performance dell'endpoint analizzato, come tempi di risposta anomali o possibili ottimizzazioni.

Di seguito sono riportati alcuni esempi di risposte ottenute dall'Agent:

Nel primo esempio (4.9), l'analisi dei log ha evidenziato un errore nel connettore utilizzato dal sistema. In particolare, l'output suggerisce di focalizzare l'attenzione sulla classe `ServiceImpl`, dove il connettore viene richiamato. Questo permette agli sviluppatori di individuare rapidamente il punto critico e avviare un'analisi più approfondita per risolvere il problema.

```
{
    "analysis": "The JSON log indicates that there is indeed
       an error. Specifically, the error mentions:
    "ERROR: Http server error while calling the connector
       policyInfoConnector (http status 500
       INTERNAL_SERVER_ERROR)"
    This indicates that there was an internal server error (
       HTTP status 500) while calling the connector:
       policyInfoConnector. Looking closely shows that there
       is a problem involving the flowServiceImpl.policyInfo
       method. To solve this issue, you should investigate
       further on the server side, especially around the
       policyInfoConnector endpoint, flowServiceImpl.
       policyInfo(). To do so, we eventually need to debug
       the error using detailed server logs, configurations
       or even the actual source code."
}
```

Listing 4.9: Esempio risposta1

Nel secondo esempio (4.10), pur non essendoci errori HTTP, l'analisi evidenzia un errore logico nell'elaborazione della richiesta: alcuni campi dell'input non rispettano i requisiti minimi, portando a un esito negativo dell'operazione (`"esito": "KO"`). L'Agent fornisce quindi un elenco dettagliato delle anomalie rilevate.

```
{
    "analysis": "In the log you provided, the errors that
       were found are:
    1. "Capitale richiesto inferiore al minimo richiedibile",
        with errorCode "KO_AMMISSIBILITA". This error was
       repeated four times in the response under the "
       returnMessages" field.
    2. "Eta in ingresso non ammessa", with errorCode "
       KO_AMMISSIBILITA".
    3. "Eta scadenza non ammessa", with errorCode "
       KO_AMMISSIBILITA".
    The response "esito" is also "KO", which means that the
       operation was not successful.
    In all of these cases, it seems that the input passed in
       the requests is not valid, or does not meet certain
       boundaries or prerequisites. To fix these errors, you
       would have to make sure that the inputs provided in
       the requests are correct. Check the requested capital,
        the input age and the expiration age in the requests.
        Make sure that the requested capital is equal to or
       greater than the minimum required, and that the input
       and expiration ages are allowed values in the system."
}
```

Listing 4.10: Esempio risposta 2

Nel terzo scenario (4.11), l'analisi non segnala errori evidenti, ma rileva un'anomalia nelle performance: il tempo di risposta dell'endpoint è superiore alla norma (`19359 ms`). Sebbene il sistema funzioni correttamente, l'Agent suggerisce di monitorare il comportamento dell'API per prevenire possibili criticità future.

```
{
    "analysis": "This JSON log is quite complex, but it seems
        that there is no obvious error. The two logs depicted
        in this JSON both report successful HTTP POST
       requests to the ‘/policy/setupPolicy‘ endpoint. In
       detail:
    1. The request had the RequestId of ‘0123456-789c-10c1-
       ba11-9b9fde9a9999‘. Headers and the body of the
       request are also reported. Among headers, many are
       related to tracing and others to the application and
       its data.
```

```
    2. The response to this request is a 200 status,
       confirming that the request has succeeded. Also, the
       response headers are reported and appear to be fine.
       They mostly consist of several ‘Vary‘ headers used to
       specify which request headers a cache mechanism should
        take into account when deciding whether to serve from
        cache.One thing to pay attention to is the time
       elapsed, reported as 19359 ms. Dpending on the nature
       of the endpoint and the application, this may be
       considered quite high. Overall, based on this log,
       nothing seems to be critical or wrong, just few
       observations for performance."
}
```

Listing 4.11: Response corretta

Infine, nel quarto esempio (4.12), l'Agent conferma l'assenza di errori nel periodo di tempo specificato, riportando una risposta senza errori.

```
{
    "analysis": "The logs provided do not seem to contain any
        explicit error or fault messages. The responses
       recorded in the logs suggest that the HTTP requests to
        the '/policy/setupPolicy' endpoint were successfully
       processed (with HTTP Status 200). Features of a
       successful request from the logs:- request method is
       POST- URI is "/policy/setupPolicy"- HTTP status code
       is 200, which means the request was successful- No
       explicit error messages in the logged information."
}
```

Listing 4.12: Risposta senza errori

Questi esempi dimostrano come l'AI Agent possa restituire analisi differenziate a seconda del contesto: alcune chiamate evidenziano errori specifici, mentre altre confermano il corretto funzionamento del sistema. In ogni caso, l'integrazione di questo strumento automatizzato ha notevolmente facilitato il processo di troubleshooting, consentendo di individuare e risolvere rapidamente anomalie nel sistema.

4.6 Considerazioni

In sintesi, il testing in Spring Boot rappresenta un elemento cardine per lo sviluppo di applicazioni robuste e affidabili. L'adozione di unit test e integration test, supportata dall'uso strategico dei mock, permette di isolare e correggere rapidamente anomalie, garantendo una maggiore copertura del codice e riducendo il debito tecnico. L'integrazione con strumenti avanzati e una pipeline CI/CD automatizzata assicura un processo di sviluppo continuo e di alta qualità, fondamentale in contesti critici come quelli della bancassurance. L'uso di AI Agent in futuro potrà velocizzare il lavoro dei programmatori fornendo supporto sempre maggiore.

Capitolo 5

Conclusioni e sviluppi futuri

Il progetto sviluppato ha rappresentato un importante passo avanti nell'implementazione di un sistema integrato per il settore bancassurance, fondato su un'architettura a microservizi che sfrutta tecnologie innovative come Spring Boot, API REST, connettori Kafka e strumenti basati su intelligenza artificiale. L'approccio adottato ha permesso di migliorare significativamente la qualità, l'affidabilità e la manutenibilità del software garantendo contemporaneamente una forte modularità e scalabilità. In particolare, la gestione del flusso di vendita attraverso State Machine e l'integrazione di un AI Agent per il troubleshooting hanno semplificato e reso più robusto il ciclo di vita del prodotto, garantendo un flusso informativo coerente e sicuro tra i vari livelli (Front End, FE2BE e Back End). L'uso di pipeline CI/CD ha permesso l'automatizzazione del processo di build, test e deployment riducendo i tempi di rilascio, minimizzando il rischio di errori in produzione e assicurando una costante validazione delle modifiche apportate. L'adozione di strumenti avanzati per il monitoraggio e l'analisi dei log, quali Splunk, ha facilitato il rilevamento tempestivo delle anomalie, migliorando la capacità di intervento e la stabilità complessiva del sistema. Questi risultati evidenziano come l'integrazione di tecnologie all'avanguardia e metodologie di sviluppo moderne possa non solo ottimizzare i processi interni, ma anche elevare gli standard qualitativi e di sicurezza in un settore altamente competitivo come quello bancario-assicurativo.

Nonostante i risultati ottenuti, il progetto presenta ancora numerosi margini di miglioramento e opportunità di evolvere con nuove tecnologie. Una delle direzioni principali riguarda l'espansione delle funzionalità dell'AI Agent, che potrebbe essere potenziato integrando modelli di intelligenza artificiale ancora più sofisticati e algoritmi predittivi capaci di anticipare anomalie e ottimizzare i processi di troubleshooting. Si potrebbe ampliare l'uso dei test automatizzati, includendo test end-to-end, di performance e di sicurezza, per

garantire una copertura più completa e una robustezza ancora maggiore del sistema. Contemporaneamente l'architettura potrà essere estesa a supportare nuovi canali e dispositivi, migliorando l'interoperabilità e l'esperienza utente attraverso Front End ottimizzati per diverse piattaforme. Un altro ambito di sviluppo potrebbe interessare l'ottimizzazione dell'orchestrazione e delle coreografie funzionali dei microservizi, in modo da gestire in maniera ancora più efficiente le comunicazioni asincrone e la scalabilità del sistema in presenza di elevati flussi di utilizzo. Infine, l'integrazione con ulteriori strumenti di monitoraggio e analisi dei dati permetterà di ottenere una visibilità ancora maggiore sulle performance dell'applicazione, facilitando l'adozione di strategie di alert e automazione dei processi di rollback in caso di malfunzionamenti. Tutti questi sviluppi porterebbero a rendere il progetto più solido e innovativo, rispondendo meglio alle esigenze del settore bancario-assicurativo e a un mercato competitivo in continua evoluzione.

Bibliografia

[1] Sara Alspaugh et al. «Analyzing log analysis: An empirical study of user log mining». In: *28th Large Installation System Administration Conference (LISA14)*. 2014, pp. 62–77.

[2] Asana. *Waterfall Project Management Methodology*. 2025. URL: https://asana.com/it/resources/waterfall-project-management-methodology (visitato il giorno 20/01/2025).

[3] Atlassian. *Introduction to Project Management*. 2025. URL: https://www.atlassian.com/agile/project-management/project-management-intro (visitato il giorno 20/01/2025).

[4] Baeldung. *Spring Boot Architecture Screenshot*. 2025. URL: https://www.baeldung.com/wp-content/uploads/2019/10/Screenshot-2019-10-31-at-22.27.41.png (visitato il giorno 20/01/2025).

[5] Inspired Brilliance. *Patterns for Microservices*. 2025. URL: https://medium.com/inspiredbrilliance/patterns-for-microservices-e57a2d71ff9e (visitato il giorno 04/01/2025).

[6] John Carnell e Illary Huaylupo Sánchez. *Spring microservices in action*. Simon e Schuster, 2021.

[7] Spring Framework Documentation. *Rest API*. 2025. URL: https://cdn-ajfbi.nitrocdn.com/GuYcnotRkcKfJXshTEEKnCZTOtUwxDnm/assets/images/optimized/rev-a2eb5e6/www.astera.com/wp-content/uploads/2020/01/rest.png (visitato il giorno 12/02/2025).

[8] Spring Framework Documentation. *Spring MVC Diagram*. 2025. URL: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/images/mvc.png (visitato il giorno 11/01/2025).

[9] Way to Easy Learn. *Microservices Communications*. 2025. URL: https://waytoeasylearn.com/learn/microservices-communications/ (visitato il giorno 05/02/2025).

[10] Imgur. *Image: LtWx95v*. 2025. URL: https://i.imgur.com/LtWx95v.png (visitato il giorno 05/01/2025).

[11] Hashmap Inc. *The What, Why, and How of a Microservices Architecture*. 2025. URL: https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9 (visitato il giorno 04/01/2025).

[12] Alan Megargel, Christopher M. Poskitt e Venky Shankararaman. «Microservices Orchestration vs. Choreography: A Decision Framework». In: *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*. 2021, pp. 134–141. DOI: 10.1109/EDOC52215.2021.00024.

[13] Glenford J Myers, Corey Sandler e Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[14] Badisa Naveen et al. «Efficient automation of web application development and deployment using jenkins: A comprehensive CI/CD pipeline for enhanced productivity and quality». In: *2023 International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*. IEEE. 2023, pp. 751–756.

[15] RadixWeb. *Waterfall vs Agile: Which Methodology is Right for Your Project?* 2025. URL: https://radixweb.com/blog/waterfall-vs-agile (visitato il giorno 10/01/2025).

[16] Publication on researchgate. *Time comparator*. 2025. URL: https://www.researchgate.net/publication/312566917/figure/fig2/AS:452632142716929@1484927320140/REST-and-SOAP-response-time-measurements-10.png (visitato il giorno 05/02/2025).

[17] Amazon Web Services. *Microservices on AWS*. 2025. URL: https://aws.amazon.com/it/microservices/ (visitato il giorno 09/01/2025).

[18] Amazon Web Services. *What is a RESTful API?* 2025. URL: https://aws.amazon.com/what-is/restful-api/ (visitato il giorno 02/01/2025).

[19] Anshu Soni e Virender Ranga. «API features individualizing of web services: REST and SOAP». In: *International Journal of Innovative Technology and Exploring Engineering* 8.9 (2019), pp. 664–671.

[20] Davide Spadini et al. «Mock objects for testing java systems: Why and how developers use them, and how they evolve». In: *Empirical Software Engineering* 24 (2019), pp. 1461–1498.

[21] Spring.io. *Spring Boot Project Page*. 2025. URL: https://spring.io/projects/spring-boot (visitato il giorno 09/01/2025).

[22] Spring.io. *Spring Data Project Page*. 2025. URL: https://spring.io/projects/spring-data (visitato il giorno 10/01/2025).

[23] The New Stack. *Kafka Architecture*. 2025. URL: https://cdn.thenewstack.io/media/2017/02/5648a9e9-kafka-arch.png (visitato il giorno 20/01/2025).