Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Informatica (Computer Engineering)

A.a. 2025/2026

Sessione di laurea Ottobre 2025

Progettazione e implementazione di un backend-for-frontend a supporto della digitalizzazione dei processi di allevamento della filiera avicola

Relatori:

Giovanni Malnati

Candidati:

Simone Paleino

Ringrazio mia madre, mio padre, mia sorella e tutti i miei parenti.

Un ringraziamento speciale al professor Malnati,
che mi ha offerto l'opportunità di intraprendere questo percorso.

Un grazie sincero ai miei amici e alla mia ragazza Chiara, che mi sono rimasti vicino durante tutta la mia esperienza accademica.

Infine, un ringraziamento ai miei colleghi di Tonicminds,
che mi hanno supportato e affiancato nello sviluppo della tesi.

Abstract

Negli ultimi anni la crescente complessità delle filiere agroalimentari ha reso indispensabile l'adozione di strumenti digitali in grado di garantire efficienza, tracciabilità e sostenibilità. La presente tesi si colloca in questo scenario, proponendo un caso studio sviluppato in collaborazione con l'azienda Wiseside e finalizzato alla digitalizzazione di un processo operativo nella filiera avicola di Martini Alimentare S.r.l.

Nell'ambito della tesi ho sviluppato un backend, che opera all'interno di un cluster Kubernetes, coordinando l'esecuzione dei diversi microservizi necessari alla digitalizzazione del processo di gestione degli allevamenti avicoli. Questa esperienza mi ha permesso di approfondire l'uso del framework Spring Boot e del linguaggio Kotlin, acquisendo conoscenze pratiche sulla gestione delle interazioni tra i microservizi in un ambiente scalabile.

Un contributo centrale della tesi riguarda la progettazione e lo sviluppo di un livello di integrazione a supporto dell'interfaccia utente, concepito come proxy server sicuro e scalabile. Questo componente ha il compito di semplificare e standardizzare la comunicazione tra frontend e motore di esecuzione del processo, aggregare e validare le richieste degli utenti, orchestrare le interazioni con i microservizi e gestire i flussi di autenticazione e autorizzazione tramite il protocollo OAuth2.x/OIDC appoggiandosi a un servizio di gestione dell'identità. La sua introduzione ha permesso di rendere più fluida l'esperienza utente e di aumentare la robustezza, la sicurezza e la manutenibilità dell'intera piattaforma.

I risultati del progetto dimostrano come l'integrazione di strumenti di Business Process Automation in un settore tradizionale come quello agroalimentare possa supportare l'innovazione, migliorare l'efficienza operativa e promuovere la sostenibilità. La ricerca conferma inoltre la validità di Camunda come soluzione solida e versatile per la trasformazione digitale di supply chain complesse, e sottolinea l'importanza di introdurre livelli di integrazione a supporto dell'interfaccia utente per garantire un reale valore aggiunto ai sistemi aziendali.

Sommario

Le attuali filiere agroalimentari evidenziano una crescente complessità di gestione. Questa evoluzione richiede, di conseguenza, l'adozione di soluzioni digitali avanzate. Tali strumenti si rivelano, infatti, indispensabili per garantire la massima qualità di ogni prodotto, consentendone il monitoraggio a partire dall'allevamento, fino ad arrivare alla tavola del consumatore finale. In questo modo, è possibile massimizzare l'efficienza operativa durante tutte le fasi del processo. Ogni step, dalla produzione alla distribuzione, viene così ottimizzato per garantire l'efficacia e l'efficienza del sistema, contribuendo inoltre al conseguimento di obiettivi di sostenibilità ambientale e sociale. L'innovazione tecnologica rappresenta quindi una risposta adeguata alle sfide emergenti nel settore, attraverso la quale è possibile garantire un elevato livello di sicurezza per i consumatori, valorizzando il lavoro degli agricoltori. Si prenda ad esempio la filiera del grano, dove ogni lotto deve essere adeguatamente identificato o, in alternativa, si consideri l'impiego di controlli rigorosi sulla temperatura durante la fase di distribuzione del latte. Tali sistemi digitali sono in grado di registrare dati con elevata precisione e accuratezza, favorendo la presa di decisioni informate, con l'obiettivo di garantire un prodotto conforme agli standard di sicurezza. In questo modo vengono inoltre promosse pratiche agricole che si pongono in una prospettiva di maggiore attenzione verso il consumatore e l'ambiente. È possibile osservare come la digitalizzazione oggigiorno si configuri come un requisito imprescindibile per consentire la gestione di un settore in perenne crescita ed evoluzione.

Il sistema che verrà descritto in questa tesi si propone di rispondere a questa esigenza, fornendo un esempio concreto di business case. Di seguito si presenta il lavoro svolto in collaborazione con l'azienda Wiseside per la digitalizzazione di un processo operativo nella filiera avicola di Martini Alimentare S.r.l., attraverso l'adozione della piattaforma Camunda. Il progetto ha come obiettivi la modellazione e l'implementazione di un sistema di gestione dei processi aziendali (BPMN) finalizzato a monitorare eventi chiave, integrare dati provenienti da dispositivi IoT e sistemi legacy, ottimizzare la raccolta, l'analisi e la visualizzazione delle informazioni lungo l'intera catena produttiva.

È stato inoltre introdotto un modello digitale della filiera. Questo strumento

avanzato raccoglie dati cruciali durante il processo produttivo, grazie a sensori e sistemi gestionali, consentendone un monitoraggio accurato. Vengono così analizzati il benessere degli animali, la qualità dei prodotti finali e l'impatto ambientale delle attività. Il modello digitale della filiera raccoglie queste informazioni, segnalando rapidamente eventuali deviazioni dalla norma e consentendo in questo modo interventi tempestivi. L'analisi dei consumi consente poi un'allocazione ottimale delle risorse, volta a ridurre sprechi, nel rispetto delle pratiche ambientali. Per permettere l'implementazione di tale modello digitale, si è scelto di adottare un orchestratore di processi e, dopo aver preso in considerazione diverse alternative, si è optato per Camunda. Questo strumento open-source consente infatti di sviluppare in tempi relativamente brevi la modellizzazione del modello digitale della filiera. Di seguito mostreremo come Camunda possa essere efficacemente integrata in architetture moderne per supportare la trasformazione digitale della supply chain agroalimentare.

Indice

Glossario

BPA (Business Process Automation) Insieme di tecniche e strumenti utilizzati per automatizzare i processi aziendali, riducendo l'intervento manuale e migliorando l'efficienza operativa. [1]

BPMN, CMMN e DMN Business Process Model and Notation (BPMN) è lo standard per la modellazione grafica dei processi aziendali. [1] Case Management Model and Notation (CMMN) è lo standard per rappresentare processi meno strutturati e basati su casi. Decision Model and Notation (DMN) è lo standard per la modellazione delle regole e decisioni aziendali. [1]

AI (Artificial Intelligence) Informatica che studia e sviluppa sistemi in grado di simulare l'intelligenza umana, come apprendimento, ragionamento e riconoscimento di pattern.

RPA (Robotic Process Automation) Tecnologia che utilizza software "bot" per automatizzare compiti ripetitivi tipicamente eseguiti da esseri umani su sistemi informatici.

Kafka Piattaforma open-source, sviluppata da Apache, per il data streaming distribuita che permette di pubblicare, sottoscrivere, archiviare ed elaborare flussi di record in tempo reale.

RabbitMQ Message broker open-source che implementa il protocollo AMQP (Advanced Message Queuing Protocol), utile per gestire code di messaggi e comunicazione asincrona tra sistemi. [2]

GIT Sistema di controllo di versione distribuito, progettato per tracciare le modifiche al codice sorgente e favorire la collaborazione tra sviluppatori [3].

CI e CD (Continuous Integration e Continuous Deployment/Delivery) Pratiche DevOps che prevedono l'integrazione continua del codice in un repository condiviso (CI) e la distribuzione automatizzata delle applicazioni in ambienti di test o produzione (CD). [4]

JPA (Java Persistence API) Specifiche Java che consentono la gestione della persistenza dei dati tramite ORM (Object-Relational Mapping), facilitando l'interazione con database relazionali. [5]

CRUD (Create, Read, Update, Delete) Acronimo che rappresenta le quattro operazioni fondamentali per la gestione dei dati in un database o in un sistema software.

MVC (Model–View–Controller) Pattern architetturale che separa la logica di un applicazione in tre componenti: modello (dati e logica), vista (interfaccia utente) e controller (gestione input).

JSON (JavaScript Object Notation) Formato leggero per lo scambio di dati, leggibile sia dall'uomo che dalle macchine, basato su una sintassi derivata da JavaScript. [6]

XML (eXtensible Markup Language) Linguaggio di markup che permette di strutturare, archiviare e trasportare dati in modo leggibile e indipendente dalla piattaforma. [7]

RESTful e REST REST (Representational State Transfer) è uno stile architetturale per la progettazione di servizi web. Un'API RESTful è un'implementazione conforme ai principi REST, che utilizza protocolli standard come HTTP.

CSRF (Cross-Site Request Forgery) Tipo di attacco informatico che induce un utente autenticato a eseguire azioni indesiderate su un'applicazione web senza il suo consenso esplicito. [8]

JWT (JSON Web Token) Standard aperto (RFC 7519) che definisce un formato compatto e sicuro per trasmettere informazioni come JSON, firmate digitalmente per garantire autenticità e integrità. [9]

ISSUE, FORK, PULL REQUEST Issue: strumento per segnalare bug, proporre nuove funzionalità o discutere miglioramenti. Fork: copia di un repository GitHub in cui si possono apportare modifiche indipendenti. Pull Request: richiesta di integrazione delle modifiche da un fork o da un branch verso il repository principale. [10]

Capitolo 1

Introduzione

Negli ultimi anni, la trasformazione digitale delle filiere agroalimentari ha assunto un ruolo sempre più centrale nell'affrontare le sfide legate alla tracciabilità, alla sostenibilità ambientale e all'efficienza operativa. In particolare, l'integrazione di strumenti software per la modellazione e l'automazione dei processi aziendali consente una gestione più trasparente, flessibile e controllabile delle attività produttive.

La seguente tesi si colloca all'interno di un progetto realizzato in collaborazione con l'azienda *Wiseside*, con l'obiettivo di digitalizzare una parte della filiera avicola dell'azienda *Martini Alimentare S.r.l.*, mediante l'uso della piattaforma *Camunda* e altri strumenti software integrati. I risultati qui presentati sono frutto della collaborazione di un team, il quale si è occupato dello sviluppo di molteplici elementi quali: il frontend, la modellazione e l'implementazione di un *proxy server* con funzioni di sicurezza, instradamento e gestione delle richieste.

1.1 Obiettivi

Questo studio si propone di esplorare la fattibilità della rappresentazione digitale di un processo operativo reale all'interno della filiera agroalimentare avicola. L'obbiettivo primario consiste nel dimostrare quanto sia possibile effettuare una transizione fedele e funzionale di tale processo virtuale. La digitalizzazione viene eseguita mediante l'utilizzo strategico di tecnologie a licenza open-source. Tali soluzioni sono caratterizzate da una grande flessibilità e accessibilità e permettono di creare sistemi robusti senza vincoli quali una licenza proprietaria.

Anche i microservizi saranno alla base dell'architettura del sistema digitale. Per realizzare le funzionalità del processo avicolo, l'applicazione viene suddivisa in unità di servizio più piccole e abbastanza indipendenti. Le funzionalità come la tracciabilità delle uova, la gestione dei mangimi e il monitoraggio delle condizioni

ambientali possono essere eseguite aggregando e orchestrando più microservizi specializzati, supportati da servizi trasversali per la sicurezza, l'integrazione e il monitoraggio. Lo sviluppo, la manutenzione e l'evoluzione del sistema sono facilitati da questa modularità e collaborazione dei componenti. Una scelta metodologica che mira a garantire scalabilità ed efficienza è la combinazione di tecnologie open source e un'architettura a microservizi.

L'obiettivo è validare questo modello attraverso un'applicazione pratica. Ad esempio, si potrà modellare il percorso di un pulcino dall'incubazione alla crescita, registrando digitalmente ogni fase. Dati come l'alimentazione, i trattamenti veterinari e le variazioni di peso potrebbero essere gestiti da microservizi distinti. Questi servizi comunicheranno poi tra loro per fornire una visione d'insieme del processo.

Sebbene il progetto non sia stato finora messo in produzione, la demo sviluppata riproduce fedelmente scenari realistici, offrendo un contributo utile alla valutazione, sia tecnica che concettuale, dell'approccio proposto.

Capitolo 2

Camunda: motivazioni e vantaggi

2.1 Introduzione

Negli ultimi anni, la piattaforma BPA è diventata un elemento chiave per le imprese che intendono migliorare l'efficienza operativa, ridurre i costi e garantire una maggiore trasparenza nei flussi di lavoro. Attualmente sono disponibili in commercio numerose piattaforme che permettono di modellare, eseguire e monitorare processi aziendali tramite lo standard BPMN 2.0, ma non tutte offrono lo stesso livello di flessibilità, scalabilità e integrazione tecnica.

Tra le soluzioni disponibili, **Camunda** si distingue per la sua architettura open-source, l'integrazione nativa con il mondo Java/Kotlin, tramite Spring Boot, e un approccio "developer-friendly", che privilegia la precisione e l'eseguibilità del modello. In questo capitolo, analizzeremo i motivi principali per cui Camunda rappresenta una scelta strategica rispetto ai suoi principali concorrenti.

2.2 Panoramica dei principali concorrenti

Di seguito, una sintesi delle piattaforme BPA più note confrontate con Camunda:

- **ProcessMaker**: la seguente piattaforma low-code presenta una particolare attenzione per gli utenti non tecnici grazie alla sua accessibilità. Offre inoltre un sistema di process intelligence e una AI integrata in grado di estrarre informazioni da documenti, link, foto e informazioni inviate dagli utenti, favorendo flussi di lavoro più intelligenti.
- **Appian**: si posiziona come una potente soluzione low-code pensata specificamente per la creazione di applicazioni aziendali complesse. La sua architettura

robusta supporta processi aziendali sofisticati che richiedono l'integrazione con diversi sistemi esistenti. Tuttavia, la piattaforma presenta una curva di apprendimento significativa e richiede un investimento di tempo considerevole. Gli utenti devono quindi acquisire familiarità con i suoi strumenti visivi e comprenderne i paradigmi di sviluppo. Ciò può rappresentare una sfida per i team con limitata esperienza nello sviluppo low-code. Per ottenere il massimo da Appian, è essenziale una formazione adeguata.

- **Pega Systems**: è la piattaforma di automazione aziendale più famosa. Pega Systems è adatto a grandi organizzazioni perché si concentra sulla gestione dei casi e sull'RPA. La piattaforma consente la liberazione delle risorse umane per compiti a maggior valore e ne consente una gestione più efficiente all'interno dell'azienda. Ciò si traduce in flussi di lavoro complessi, processi mission-criticali e attività ripetitive. È un partner strategico per le aziende che devono affrontare sfide operative su larga scala perché offre scalabilità, robustezza e conformità.

- **Bizagi**: offre un'interfaccia che è adatta anche a coloro che non sono esperti di programmazione. Disegnare i processi senza codice consente di impostare e ottimizzare i flussi. Questo alleggerisce la responsabilità dell'IT coinvolgendo direttamente gli utenti aziendali nella definizione delle attività. Il risultato è un processo di adozione più rapido, una collaborazione più efficiente tra le divisioni e un contributo concreto ai progetti di digitalizzazione.

- **Nintex, Salesforce Flow, Pipefy, Kissflow, Microsoft Power Automate**: sono piattaforme di automazione che utilizzano interfacce visive con elementi trascinabili. Sono progettati per funzionare con processi lineari e a bassa complessità come approvazioni, notifiche e aggiornamenti di dati. Inoltre, spesso includono connettori preconfigurati per integrare applicazioni eterogenee. Mentre Pipefy e Kissflow offrono una struttura guidata per l'organizzazione dei flussi, Salesforce Flow offre un'integrazione nativa con l'ecosistema Salesforce all'interno del gruppo. Gli strumenti come questi riducono la dipendenza dai team IT e consentono iterazioni rapide nei processi.

Camunda si rivolge a un pubblico che ha esigenze di automazione più sofisticate, dove controllo, trasparenza e precisione sono fondamentali, anche se ciascuna di queste soluzioni ha i suoi vantaggi.

2.3 Pregi e vantaggi di Camunda

2.3.1 Architettura open-source e flessibile

Camunda offre un modello di business distintivo. La piattaforma è totalmente open-source, rendendo il codice sorgente accessibile a tutti. Questa caratteristica risulta fondamentale per permettere una profonda personalizzazione, consentendo integrazioni avanzate, una completa flessibilità, nei limiti della licenza di copyright. Nella sua iterazione più recente, Camunda 8, la piattaforma abbraccia un approccio cloud-native, il quale permette una più facile possibilità di condivisione con altri utenti e introduce, di conseguenza, una maggiore sicurezza nella permanenza degli schemi di flusso. A differenza di altre soluzioni di gestione dei processi aziendali come Appian o Pega, Camunda si distingue in quanto non richiede accordi restrittivi. L'adozione di queste piattaforme comporta solitamente contratti potenzialmente vincolanti per le aziende. Tali strumenti possono richiedere inoltre l'uso di ambienti proprietari, limitando le opzioni di personalizzazione e implicando possibili aumenti dei costi nel tempo.

2.3.2 Integrazione nativa con Java/Kotlin e Spring Boot

Camunda è stata concepita per adattarsi perfettamente agli ecosistemi basati su Java/Kotlin e, con l'adozione di Spring Boot, viene facilitata l'integrazione del motore BPMN, il quale può essere direttamente implementato all'interno di applicazioni monolitiche o microservizi. Questa integrazione nativa offre un elevato grado di controllo sul ciclo di vita dei processi.

Si immagini, per esempio, di voler sviluppare un sistema di gestione degli ordini: si potrà usare Spring Boot per creare un microservizio dedicato, che si occuperà della creazione e della spedizione degli ordini. All'interno di questo microservizio è possibile incorporare il motore Camunda BPMN che andrà a orchestrare la sequenza delle attività, come la ricezione dell'ordine, la verifica della disponibilità, l'elaborazione del pagamento e la notifica al cliente. La logica di business, come il controllo della disponibilità del prodotto, risiede nel codice Java/Kotlin del microservizio. Camunda BPMN definisce così il flusso generale delle operazioni. La stretta integrazione garantisce che queste due parti lavorino in armonia, in modo da avviare un processo Camunda all'arrivo di un nuovo ordine. Il motore guiderà poi l'ordine attraverso le varie fasi definite nel modello BPMN. Questa sinergia garantisce che il flusso del processo sia sempre in linea con le esigenze operative.

Inoltre, l'utilizzo di Spring Boot semplifica notevolmente la configurazione e l'avvio del motore Camunda all'interno dell'applicazione Java/Kotlin. Non sono di fatto necessarie configurazioni esterne complesse, in quanto il motore diventa parte

integrante dell'applicazione, portando a ridurre la complessità dell'architettura e favorendo anche una maggiore manutenibilità.

La capacità di gestire il ciclo di vita del processo in modo molto granulare è un'altra caratteristica significativa. Gli sviluppatori possono osservare ogni fase del processo e correggere eventuali problemi. Per garantire che i sistemi aziendali funzionino correttamente, è fondamentale avere questo livello di controllo e visibilità. La coerenza e, di conseguenza, una netta distinzione delle responsabilità sono garantite dalla completa integrazione del modello che definisce il flusso e del codice che esegue la logica. Mentre il modello BPMN definisce "cosa" deve accadere, il codice Java/Kotlin implementa "come" per ogni passaggio automatizzato.

2.3.3 Eseguibilità dei modelli BPMN

Molte piattaforme mostrano solo un diagramma dei processi, ma Camunda va oltre. Qui, il modello BPMN non è solo una rappresentazione visiva, ma funziona davvero. Quando si costruisce un processo, non è più necessario scrivere un codice separato per farlo funzionare. In passato, trasformare un diagramma in un sistema operativo richiedeva tempo e spesso introduceva errori. Con Camunda, invece, tutto avviene all'interno dello stesso modello e il legame tra progetto e sistema diventa immediato.

Immaginiamo di dover gestire l'approvazione di una fattura. Nel diagramma, l'invio della notifica all'approvatore non è un dettaglio da aggiungere successivamente, ma è già previsto e fa parte integrante del modello. Ciò significa che ciò che si vede sul diagramma corrisponde esattamente a ciò che il sistema farà. Non ci sono passaggi intermedi che possano generare fraintendimenti.

Invece, nei metodi tradizionali, un analista crea il diagramma e poi uno sviluppatore lo traduce in codice. Molte volte le intenzioni originali vengono interpretate in modo diverso, con la possibilità di commettere errori. Tuttavia, il modello stesso è eseguibile con Camunda. Ogni modifica al diagramma riflette immediatamente il comportamento del sistema, rendendo più facile aggiornare i processi e migliorare il flusso di lavoro.

In pratica, questo metodo rende l'intero ciclo affidabile e più veloce. Il modello è il motore del processo, non solo una linea d'azione da seguire. Pertanto, le persone che lavorano sui processi possono concentrarsi sulle decisioni e sulle regole aziendali senza doversi preoccupare di tradurre i diagrammi in codice o di correggere gli errori che si verificano durante la fase di implementazione.

2.3.4 Capacità di scaling e supportare ambienti complessi

Camunda 8 introduce Zeebe, un motore di flusso di lavoro distribuito e orientato agli eventi. Questa architettura è stata progettata per gestire un elevato volume di

traffico. Inoltre, questa architettura è in grado di supportare anche gli ambienti a microservizi su Kubernetes e la sua struttura cloud-native consente a Camunda di affrontare scenari complessi, i quali richiedono un'elevata scalabilità, resilienza e capacità di parallelismo. Camunda 8 è in competizione con piattaforme come Pega; a differenza di quest'ultima, offre però una maggiore apertura tecnologica. Zeebe, infatti, processa gli eventi in modo efficiente, in quanto ogni evento rappresenta un cambiamento di stato. Non ci sono punti di fallimento singoli, questo perché Kubernetes orchestra i container di Zeebe garantendo la scalabilità automatica e il recupero dagli errori. Questo aspetto è cruciale per i sistemi mission-critical. Si pensi, per esempio, alla gestione degli ordini in un e-commerce, dove ogni ordine è un flusso di eventi. Zeebe è in grado di gestire migliaia di ordini simultaneamente e ogni ordine è in grado di attivare diverse azioni, come l'invio di email, l'aggiornamento del database o la chiamata ad altri servizi. La scalabilità di Zeebe permette così di gestire picchi di traffico come, ad esempio, quelli causati dalle promozioni speciali durante il Black Friday. La resilienza del sistema garantisce che i flussi di lavoro non si interrompano, anche nel caso in cui un nodo del cluster dovesse fallire. In una tale evenienza, Zeebe riavvierebbe automaticamente le operazioni, minimizzando di conseguenza i tempi di inattività.

2.3.5 Controllo completo e osservabilità

Operate e Tasklist sono strumenti avanzati di Camunda che migliorano la visibilità dei processi aziendali.

Operate consente di tenere d'occhio l'esecuzione dei processi in tempo reale. In questo modo è possibile visualizzare quali istanze di processo sono in corso e trovare i colli di bottiglia nelle operazioni. Ciò fornisce una panoramica dell'attuale situazione.

Invece, Tasklist gestisce le attività assegnate agli utenti. Ciò consente ai lavoratori di vedere e completare le loro attività, rendendo il flusso di lavoro più semplice. Questo chiarisce chi è responsabile di cosa.

Rispetto a molte piattaforme low-code disponibili sul mercato, questi strumenti offrono un livello di osservabilità molto alto. Spesso, queste soluzioni utilizzano interfacce molto semplificate e nascondono la logica fondamentale. Invece, Camunda mostra i dettagli operativi, che sono una caratteristica essenziale della buona gestione. Ciò consente di comprendere meglio come funzionano i processi.

2.3.6 Assenza lock-in tecnologico

Camunda offre una notevole flessibilità in termini di integrazione grazie al fatto che non impone l'adozione di un unico ambiente di sviluppo o di esecuzione. Questa scelta strategica consente agli utenti di orchestrare i processi aziendali attraverso

molteplici canali tecnologici. Per esempio, è possibile avviare e gestire i processi direttamente tramite API REST. Ciò apre le porte a integrazioni dinamiche con sistemi esterni.

L'integrazione con broker di messaggi come Kafka o RabbitMQ è diretta e semplice, anzi sono progettati per gestire la comunicazione con le applicazioni distribuite. Questo è un aspetto fondamentale quando i sistemi devono scambiarsi informazioni senza rispondere in tempo reale. Ad esempio, un Camunda può ricevere eventi da Kafka o inviare messaggi a una coda che è strettamente legata a RabbitMQ.

La piattaforma si distingue per l'apertura verso tecnologie differenti: rimangono da un canto soluzioni come Appian o Salesforce Flow che costringono l'utente ad utilizzare il proprio ecosistema imponendo l'uso di componenti esclusivamente proprietari, con conseguente restringimento delle possibilità di integrazione con altri strumenti. Camunda, viceversa, è caratterizzato da una architettura aperta che sembra dettata a incoraggiare la libera scelta. Si progetta così, con facilità, una soluzione mirata alle proprie esigenze, impiegando volutamente materiali conformi e poco soggetti al fenomeno del lock-in, e, insieme, s'annientano i costi d'integrazione.

2.3.7 Vantaggi per team tecnici e agili

Camunda è pensato per supportare i team che si ispirano a metodologie agili e pratiche DevOps, in modo che la cooperazione sia efficace e nulla ostacoli i rilasci rapidi. Il modello di processo di business, nella fattispecie, è un'effettiva creazione software che può esser inclusa in un sistema di controllo delle versioni quali Git, e che dunque riconosce ogni cambiamento, agevola la cooperazione tra i vari autori e permette di salvare o ripristinare una qualsiasi versione.

Un'altra pietra angolare della costruzione è costituita dalla testabilità. Camunda offre a chi si occupa dei modelli la possibilità di sviluppare test automatici per accertare il flusso corretto del processo così come le decisioni logiche. La ripetizione regolare dei test garantisce il corretto funzionamento dei processi adottati, riducendo il rischio di errori in produzione.

Inoltre, Camunda si integra in modo ottimale nelle pipeline di CI e CD, consentendo l'automazione del deployment dei modelli di processo. Ogni modifica approvata nel repository Git può avviare automaticamente le fasi di build, test e rilascio del modello.

Questa filosofia developer-first rappresenta un vantaggio competitivo significativo per le organizzazioni che puntano a garantire un'elevata qualità dei propri prodotti e a ridurre i tempi di rilascio. Essa consente ai team di concentrarsi sulla creazione di valore, liberandoli dai vincoli imposti da processi manuali o sistemi rigidi, e favorendo la capacità di iterare rapidamente sui processi aziendali — aspetto cruciale in contesti dinamici. In questo modo, le aziende possono rispondere con

prontezza alle mutevoli esigenze del mercato. L'adozione di Camunda in un contesto agile e DevOps promuove una cultura di miglioramento continuo, fornendo agli sviluppatori gli strumenti necessari per progettare, testare e rilasciare i processi aziendali in modo efficiente.

2.4 Quando *non* scegliere Camunda

Camunda è un tool molto potente e versatile che, tuttavia, non risulta sempre la soluzione ideale per qualsiasi organizzazione. In quei contesti in cui il team di sviluppo non gode di skill tecniche di alto livello o quando la priorità è quella di adottare un approccio completamente "no-code", potrebbe essere conveniente vagliare alternative più adatte, più facili da gestire e più rispondenti a quello specifico contesto.

Piattaforme come Bizagi, Nintex o ProcessMaker sono progettate per semplificare la creazione e l'automazione dei flussi di lavoro in quanto offrono spesso interfacce grafiche intuitive, strumenti drag-and-drop e funzionalità preconfigurate che riducono notevolmente la necessità di competenze di programmazione. Questa facilità d'uso si traduce in un periodo di acquisizione del valore (time-to-value) significativamente più breve in un team con limitata esperienza di codifica, consentendo la costruzione e l'implementazione dei processi in tempi più brevi rispetto a quanto previsto da una piattaforma con maggiori necessità di configurazioni tecniche.

Un piccolo ufficio marketing, per esempio, che necessita di automatizzare l'approvazione delle campagne pubblicitarie, potrebbe trovare in Bizagi una soluzione immediata. Grazie a un editor visuale, è possibile disegnare il flusso di lavoro, definire le fasi di revisione e assegnare i compiti ai membri del team senza scrivere una sola riga di codice, mentre con Camunda, seppur offra una personalizzazione senza pari, tale automatizzazione richiederebbe probabilmente l'intervento di sviluppatori esperti per configurare i modelli di processo e le integrazioni necessarie, aumentando i tempi e i costi correlati.

È importante sottolineare che questo vantaggio in termini di velocità e facilità di adozione può comportare una rinuncia alla flessibilità tecnica intrinseca di Camunda. Le piattaforme no-code, per loro natura, presentano delle limitazioni in termini di personalizzazione avanzata o di integrazione con sistemi molto specifici e non standardizzati. Sebbene Bizagi, Nintex o ProcessMaker offrano funzionalità di integrazione, queste potrebbero non coprire tutte le casistiche più complesse o proprietarie, mentre Camunda, al contrario, eccelle nella sua capacità di adattarsi a requisiti tecnici estremamente particolari e di orchestrare sistemi eterogenei, grazie alla sua architettura basata su standard aperti e API robuste.

Pertanto, prima di scegliere una piattaforma di orchestrazione dei processi, si devono considerare attentamente il contesto specifico, le competenze del team e gli obiettivi dell'azienda. Le soluzioni Bizagi, Nintex e ProcessMaker possono essere la scelta più strategica per progetti che richiedono rapidità di implementazione e un'interfaccia accessibile agli utenti non tecnici, anche a scapito di una minore flessibilità tecnica.

2.5 Conclusione

Camunda costituisce una scelta strategica di assoluta rilevanza per le realtà che intendono realizzare soluzioni BPMN robuste, scalabili e nativamente fruibili in ecosistemi applicativi costruiti con tecnologia Java/Kotlin; a differenza di molte tra le sue concorrenti, in particolare di quelle orientate a modelli low-code/no-code, pone infatti l'accento sulla centralità dell'esecuzione dei processi nella propria architettura, garantendo elevati livelli di affidabilità e consistenza nell'automazione.

La piattaforma si distingue per il tipo di integrazione: la sua concezione ne favorisce, infatti, un'inclusione organica all'interno delle architetture IT preesistenti, creando così le condizioni per una comunicazione chiara e netta con gli altri sistemi aziendali. La possibilità di operare in tal senso si traduce in un ampliamento dell'efficienza operativa e fa venir meno la creazione di compartimenti stagni; mentre il carattere aperto della progettazione attribuisce alla struttura un'ineccepibile duttilità, permettendo agli sviluppatori di ricalibrare e di arricchire d'opzioni il motore, secondo le particolari esigenze che ciascuna delle singole imprese possa sostenere.

Benché l'adozione di Camunda presupponga un impiego iniziale di competenze tecniche altamente specialistiche, i vantaggi a lungo raggio che ne derivano sono tali da giustificare l'investimento. A contribuire al solido funzionamento delle operazioni automatizzate è la fiducia intrinseca del motore che offre una visione completa delle sequenze operative, rendendo facile il riconoscimento di eventuali colli di bottiglia.

Le aziende che puntano a una maggiore scalabilità, a elevati standard qualitativi del software e all'automazione di flussi complessi troveranno in Camunda la soluzione ottimale. La piattaforma, che aderisce a standard aperti consolidati, garantisce prestazioni elevate e una solida architettura, rivelandosi un alleato strategico per la trasformazione digitale e l'ottimizzazione dei processi aziendali. La sua architettura aperta facilita l'integrazione con le tecnologie emergenti e supporta l'evoluzione delle esigenze aziendali, mentre la sua capacità di gestire un elevato volume di transazioni la rende ideale per gli scenari ad alta intensità di lavoro, garantendo alle organizzazioni la prontezza necessaria per rispondere rapidamente ai cambiamenti del mercato.

Capitolo 3

Architettura del Sistema e Topologia dei Servizi Software

3.1 Camunda 8: Architettura, Componenti e Evoluzione dalla Versione 8.6 alla 8.7

3.1.1 Introduzione a Camunda 8

Camunda 8 rappresenta una piattaforma avanzata per l'automazione dei processi aziendali, progettata per gestire flussi di lavoro complessi in ambienti distribuiti e scalabili. A differenza della versione precedente, Camunda 7, che si basava su un'architettura monolitica, Camunda 8 adotta un'architettura incentrata su microservizi, con Zeebe come motore di orchestrazione centrale del sistema. La piattaforma è composta da diversi componenti principali:

- **Zeebe**: motore di orchestrazione dei processi, responsabile dell'esecuzione dei flussi di lavoro definiti nei BPMN.
- **Tasklist**: interfaccia utente per la gestione delle attività assegnate agli utenti.
- **Operate**: strumento per il monitoraggio e l'analisi in tempo reale delle istanze di processo.
- **Optimize**: soluzione per l'analisi avanzata delle performance dei processi e per la generazione di reportistica.
- **Modeler**: editor grafico per la progettazione dei modelli BPMN, CMMN e DMN.

3.1.2 Zeebe: Il Motore di Orchestrazione dei Processi

Architettura Zeebe è il motore di orchestrazione principale per Camunda 8, in quanto questa tecnologia gestisce il flusso dei processi aziendali. Si articola attorno a quattro componenti distinti, ognuno dei quali svolge un ruolo centrale; tuttavia, tutti lavorano insieme per garantire un'elevata efficienza. Di seguito, i suoi componenti:

- **Client**: strumento attraverso il quale l'utente può iniziare nuove istanze di processo, trasmettere messaggi o assegnare compiti pendenti.
- **Gateway**: punto di contatto tra Zeebe e i client esterni, accessibile tramite protocolli come gRPC e HTTP.
- **Broker**: fulcro dell'architettura, gestisce i processi, smista messaggi tra i nodi e controlla l'intero flusso di lavoro.
- **Exporter**: estrae i dati dai processi in corso per inviarli a sistemi terzi, consentendo analisi e reportistica avanzata.

3.1.3 Tasklist: Gestione delle Attività Utente

L'interfaccia principale è Tasklist, che consente agli utenti di vedere, gestire e completare le attività assegnate. Le funzionalità principali includono:

- **Filtri personalizzati**: possibilità di creare e salvare filtri su misura per visualizzare solo le attività più rilevanti.
- **Priorità delle attività**: assegnazione di livelli di priorità per garantire che i compiti più urgenti vengano eseguiti per primi.
- **Mostra il diagramma di processo**: consente di visualizzare l'integrazione del diagramma BPMN per comprendere meglio il contesto del compito.
- **Supporto per la gestione dei documenti**: possibilità di allegare e gestire documenti direttamente all'interno dell'interfaccia.

3.1.4 Elasticsearch: Analisi e Monitoraggio Avanzato

Elasticsearch svolge un ruolo fondamentale all'interno dell'architettura di Camunda 8, fungendo da motore per l'analisi avanzata e il monitoraggio dei processi. Le sue caratteristiche principali includono:

- **Indicizzazione dei dati**: consente ricerche rapide e analisi dettagliate delle metriche di processo.

- **Configurazione flessibile**: può essere utilizzato all'interno del cluster Kubernetes o come servizio esterno.
- **Sicurezza avanzata**: grazie a estensioni come X-Pack, supporta monitoraggio delle attività, crittografia e controllo accessi basato sui ruoli.
- **Gestione privilegi**: configurazioni specifiche garantiscono che Camunda acceda ai dati necessari in sicurezza, evitando rischi legati a permessi troppo permissivi.

3.1.5 Confronto tra Camunda 8.6 e Camunda 8.7

La versione 8.7 di Camunda segna un progresso significativo rispetto alla 8.6. Le novità principali sono:

- **Supporto per subprocessi ad-hoc in Operate**: maggiore flessibilità nella gestione dei flussi di lavoro in contesti imprevisti.
- **Gestione avanzata dei documenti in Tasklist**: selettore di file e anteprima integrata migliorano l'esperienza d'uso e la produttività.
- **Miglioramenti nelle performance di Optimize**: algoritmi ottimizzati per generare report complessi in tempi molto più brevi.

3.1.6 Conclusioni

In sintesi, Camunda 8 dimostra di essere una piattaforma moderna, affidabile e scalabile, in grado di soddisfare esigenze complesse di automazione dei processi. La versione 8.7 rafforza ulteriormente il ruolo di Camunda come strumento strategico per la trasformazione digitale.

3.2 Spring Boot e Kotlin: Architettura, Sicurezza e Convenzioni

3.2.1 Introduzione a Spring Boot

L'obbiettivo di Spring Boot, un framework open-source basato su Java, è quello di rendere notevolmente più semplice sviluppare applicazioni stand-alone pronte per la produzione. Il concetto di convention-over-configuration è alla base di Spring Boot. Con questo metodo, gli sviluppatori possono evitare configurazioni XML lunghe e complesse, favorendo una prototipazione rapida e uno sviluppo più agile. Grazie a queste funzionalità, Spring Boot è diventato uno strumento ideale sia per

la creazione di applicazioni web che per l'implementazione di microservizi, che sono sempre più comuni nei sistemi distribuiti contemporanei.

3.2.2 Flessibilità di Spring Boot

La flessibilità di Spring Boot si manifesta in una serie di aspetti importanti, che lo rendono uno strumento potente e versatile per lo sviluppo software:

- **Configurazione automatica**: la possibilità di configurare automaticamente il progetto tramite l'annotazione `@EnableAutoConfiguration` è uno dei maggiori vantaggi di Spring Boot. In altre parole, il framework analizza le dipendenze del progetto e mette insieme i componenti essenziali senza che sia necessario un intervento manuale. Pertanto, gli sviluppatori possono concentrarsi sulla logica dell'applicazione, risparmiando tempo e riducendo lo sforzo necessario per configurare.

- **Starter POMs**: le collezioni predefinite di dipendenze starter POMs facilitano l'integrazione di librerie comunemente utilizzate, come quelle per la sicurezza, la comunicazione web e la gestione dei dati. I pacchetti consentono di incorporare tutte le funzionalità necessarie in una configurazione senza doversi preoccupare di gestire ogni dipendenza singolarmente. Questo metodo riduce la complessità del progetto e aumenta la produttività.

- **Embedded Servers**: Spring Boot supporta server incorporati come Tomcat, Jetty e Undertow. La loro caratteristica consente di eseguire un'applicazione direttamente, senza dover installare e configurare un server esterno. L'uso di server embedded semplifica il deployment, rende più agevole la fase di testing e velocizza il rilascio in produzione delle applicazioni.

- **Actuator**: uno strumento aggiuntivo molto utile che consente di gestire e monitorare endpoint specifici. Gli endpoint permettono di accedere a informazioni come lo stato di salute del sistema (*Health Checks*), le prestazioni e i dettagli di configurazione. In contesti di produzione, queste caratteristiche sono particolarmente utili perché è fondamentale controllare continuamente lo stato dell'applicazione per garantire stabilità e affidabilità.

3.2.3 Vantaggi di Kotlin rispetto a Java

Kotlin, sviluppato da JetBrains, è un linguaggio moderno con molti vantaggi mentre rimane interoperabile completamente con Java. Ciò consente ai team di introdurre gradualmente Kotlin senza dover riscrivere i progetti esistenti da zero. Significa che le aziende possono continuare ad aggiornare e modernizzare le basi di codice

passo dopo passo senza perdere l'investimento che hanno già fatto in Java. Tra i principali benefici di Kotlin si possono evidenziare:

- **Sintassi concisa**: la possibilità di ridurre il codice boilerplate, che è il codice ripetitivo e ridondante che spesso occupa molte righe in Java, è uno dei vantaggi di Kotlin. Costrutti come le `data class` creano automaticamente metodi comuni come `equals`, `hashCode` e `toString`, rendendo il codice più leggero e compatto. Le espressioni lambda e le funzioni di estensione facilitano la scrittura e la manutenzione del codice.

- **Null Safety**: uno dei problemi più frequenti nello sviluppo con Java è il `NullPointerException`. Kotlin lo risolve introducendo un sistema di tipi che distingue chiaramente tra riferimenti nullabili e non nullabili, riducendo drasticamente i bug a runtime e aumentando la stabilità delle applicazioni.

- **Coroutines**: Kotlin supporta nativamente le coroutines, uno strumento potente per la programmazione asincrona e concorrente. Consentono di scrivere codice non bloccante in modo leggibile e semplice, migliorando la reattività delle applicazioni e semplificando la gestione di operazioni che richiedono tempo (es. richieste di rete o accesso a database).

- **Compatibilità con Java**: uno dei motivi principali che giustifica l'adozione di Kotlin è la sua completa interoperabilità con Java. È possibile utilizzare librerie e framework Java direttamente in Kotlin, e viceversa, senza modifiche complesse.

3.2.4 Annotazioni di Spring Boot

I vari componenti di un'applicazione possono essere definiti, organizzati e gestiti con l'aiuto di Spring Boot. Le annotazioni principali includono:

- `@Service`: identifica una classe come contenitore della logica di business dell'applicazione. È una specializzazione di `@Component`, e consente di separare la logica di business dagli altri livelli (persistenza, presentazione).

- `@Entity`: associa una classe a una tabella del database e la definisce come entità JPA. Ogni istanza corrisponde a una riga della tabella. È usata insieme a Spring Data JPA per semplificare le operazioni CRUD.

- `@Controller` e `@RestController`: la prima viene utilizzata per i controller nel pattern MVC, mentre la seconda combina `@Controller` e `@ResponseBody` per restituire direttamente JSON o XML, semplificando la creazione di API RESTful.

- `@Repository`: estensione di `@Component` per il livello di accesso ai dati. Traduce automaticamente le eccezioni specifiche del database in eccezioni generiche di tipo `DataAccessException`, rendendo il codice più robusto e pulito.

3.2.5 Spring Security

Spring Security, un potente e flessibile modulo di gestione della sicurezza, è integrato in Spring Boot. Le funzionalità principali includono:

- **Autenticazione e Autorizzazione**: supporto a vari metodi (login form, HTTP Basic, OAuth2, LDAP), fino a soluzioni distribuite con Single Sign-On (SSO).
- **Protezione CSRF**: protezione automatica contro attacchi Cross-Site Request Forgery.
- **Gestione delle sessioni**: possibilità di regolare il comportamento delle sessioni (limite sessioni parallele, scadenza automatica).
- **Annotazioni di sicurezza**: utilizzo di annotazioni come `@PreAuthorize` e `@Secured` per controllare l'accesso a livello di metodo.

3.2.6 Conclusioni

L'integrazione di Spring Boot e Kotlin consente la creazione di applicazioni moderne, efficienti e sicure. La sintassi concisa e le funzionalità avanzate di Kotlin, unite alla flessibilità e agli strumenti di Spring Boot (annotazioni, sicurezza, server embedded), offrono un ambiente completo e produttivo per lo sviluppo di applicazioni scalabili e robuste.

3.3 Kubernetes: Autoscalabilità, Gestione delle Repliche e Allocazione dei Pod

3.3.1 Configurazione del cluster e strumenti utilizzati

Nel progetto, il cluster Kubernetes è stato inizialmente configurato utilizzando `kubeadm`, uno strumento open-source che consente di eseguire il bootstrap rapido di cluster on-premise. Comandi essenziali come `kubeadm init`, per l'inizializzazione del control plane, e `kubeadm join`, per l'aggiunta dei nodi worker, hanno permesso di predisporre un ambiente funzionante in tempi ridotti. La fase iniziale di deployment è stata condotta su un cluster locale, dedicato a scopi di sviluppo

e test. Una volta validata la configurazione, la soluzione è stata migrata su un cluster Kubernetes di produzione, fornito e gestito dal cliente **Wiseside**.

È stato scelto Helm, un package manager per Kubernetes, per supervisionare e coordinare i deployment. I diagrammi Helm consentono la definizione, l'installazione e l'aggiornamento di applicazioni containerizzate. Questi pacchetti semplificano significativamente il ciclo di vita di un'applicazione fornendo una descrizione di tutte le risorse Kubernetes necessarie. Anche l'Ingress Controller **Apache APISIX** è stato distribuito tramite un diagramma specifico, rendendo la configurazione delle regole di routing e la gestione del traffico in ingresso al cluster più semplice.

3.3.2 Concetti fondamentali di Kubernetes

Le applicazioni containerizzate di Kubernetes sono organizzate attorno alle risorse, che definiscono lo stato desiderato del sistema. Le risorse più importanti utilizzate nel progetto includono:

- **Pod**: rappresenta la più piccola unità eseguibile in Kubernetes. Un pod è composto da uno o più container strettamente collegati che condividono lo stesso namespace di rete e possono ospitare volumi di storage comuni.
- **Deployment**: è un controller che garantisce che il numero di repliche desiderato di un Pod sia sempre attivo. Supporta rollback e *rolling updates*.
- **StatefulSet**: è un controller utilizzato per la gestione di carichi di lavoro stateful che richiedono un'identità di rete stabile e storage persistente. Nel progetto, PostgreSQL è stato distribuito come StatefulSet e collegato a un PersistentVolume.
- **Ingress**: consente l'accesso esterno ai servizi interni del cluster tramite regole di routing HTTP/HTTPS. È stato gestito con Apache APISIX, che funge da API Gateway cloud-native e da Ingress Controller.

3.3.3 Servizi distribuiti su Kubernetes

Sul cluster Kubernetes sono stati distribuiti diversi servizi fondamentali per il progetto. Tra i principali:

- **Backend**: microservizio containerizzato e distribuito tramite Deployment, configurato con più repliche per garantire disponibilità e scalabilità.
- **Frontend**: applicazione web containerizzata, anch'essa distribuita tramite Deployment con repliche multiple. Esposta tramite Ingress APISIX.

- **Camunda Platform 8**: l'intera suite (Zeebe, Zeebe Gateway, Operate, Tasklist, Optimize e Identity basato su Keycloak) distribuita mediante Helm chart ufficiale.
- **API Gateway / Ingress Controller (Apache APISIX)**: punto di ingresso unificato del cluster, distribuito tramite Helm. Ha fornito routing avanzato e bilanciamento del carico.
- **Database PostgreSQL**: distribuito come StatefulSet e associato a PersistentVolume, utilizzato sia per Keycloak che per i microservizi sviluppati.

3.3.4 Fasi di distribuzione e gestione dei deployment

Il processo di deployment è stato organizzato in due fasi:

1. **Cluster locale**: creato con `kubeadm` a scopo di sviluppo e test. Sono stati verificati componenti fondamentali come Deployment, StatefulSet e Ingress.
2. **Cluster di produzione (Wiseside)**: applicazione dei manifest YAML e Helm chart validati in precedenza. Garantita scalabilità e gestione affidabile delle risorse.

3.3.5 Autoscalabilità e gestione delle repliche

Sono state implementate diverse strategie di autoscalabilità:

- **Horizontal Pod Autoscaler (HPA)**: regola dinamicamente il numero di repliche dei Pod in base alle metriche (CPU, memoria).
- **Cluster Autoscaler**: adatta automaticamente il numero di nodi disponibili in base alle esigenze di scheduling dei Pod.

3.3.6 Allocazione dei Pod e scheduling

Lo scheduler di Kubernetes valuta una serie di criteri per posizionare i Pod:

- risorse richieste (CPU, memoria, storage);
- affinità e anti-affinità tra Pod e nodi;
- taints e tollerations;
- priorità del Pod e configurazioni QoS.

3.3.7 Esempio di manifest YAML per Keycloak

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: keycloak
  labels:
    app: keycloak
spec:
  replicas: 2
  selector:
    matchLabels:
      app: keycloak
  template:
    metadata:
      labels:
        app: keycloak
    spec:
      containers:
      - name: keycloak
        image: quay.io/keycloak/keycloak:21.1.1
        ports:
        - containerPort: 8080
        env:
        - name: KEYCLOAK_ADMIN
          value: "admin"
        - name: KEYCLOAK_ADMIN_PASSWORD
          value: "adminpassword"
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
          limits:
            memory: "1Gi"
            cpu: "1"
```

3.3.8 Esempio di Helm Chart per Keycloak

```
# Chart.yaml
apiVersion: v2
name: keycloak-chart
version: 0.1.0
```

```
appVersion: "21.1.1"

# values.yaml
replicaCount: 2
image:
  repository: quay.io/keycloak/keycloak
  tag: "21.1.1"
  pullPolicy: IfNotPresent
service:
  type: ClusterIP
  port: 8080
adminUser: admin
adminPassword: adminpassword

# templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "keycloak-chart.fullname" . }}
  labels:
    app: {{ include "keycloak-chart.name" . }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ include "keycloak-chart.name" . }}
  template:
    metadata:
      labels:
        app: {{ include "keycloak-chart.name" . }}
    spec:
      containers:
      - name: {{ .Chart.Name }}
        image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
        ports:
        - containerPort: {{ .Values.service.port }}
        env:
        - name: KEYCLOAK_ADMIN
          value: {{ .Values.adminUser }}
        - name: KEYCLOAK_ADMIN_PASSWORD
          value: {{ .Values.adminPassword }}
```

3.4 Firebase Cloud Messaging: gestione di messaggi e notifiche

3.4.1 Panoramica generale

Firebase Cloud Messaging (FCM) è un servizio gratuito offerto da Google che consente la consegna affidabile di messaggi e notifiche su piattaforme Android, iOS e web. Grazie alla sua architettura scalabile, rappresenta una delle soluzioni più diffuse per implementare comunicazioni push moderne ed efficienti. Permette la gestione di notifiche e pacchetti dati (payload) fino a 4 KB, risultando adatto sia per inviare semplici avvisi all'utente sia per distribuire informazioni più articolate elaborate direttamente dall'applicazione.

3.4.2 Perché adottare Firebase per notifiche e messaggistica

Firebase Cloud Messaging ha molti vantaggi:

- **Cross-platform e gratuito**: utilizzabile senza limiti di invio e senza costi di licenza. Supporta nativamente Android, iOS e web, semplificando lo sviluppo multi-piattaforma.
- **Affidabilità ed efficienza**: consumo ridotto di energia sui dispositivi mobili e consegna affidabile anche in condizioni difficili (app in background o chiusa).
- **Targeting avanzato**: invio a singoli dispositivi tramite token, a gruppi o a utenti iscritti a specifici topic. Con Firebase Analytics è possibile segmentare dinamicamente gli utenti.
- **Payload flessibili**: distinzione tra messaggi di notifica (gestiti dal sistema operativo) e messaggi di dati (fino a 4 KB, gestiti dall'app). Supporto a opzioni avanzate come TTL, collapse key, suoni, immagini e tracciamento conversioni.
- **Integrazione con Firebase Console**: invii programmati o immediati, test A/B e configurazione delle campagne senza scrivere codice.

3.4.3 Architettura di invio e ricezione

Un'implementazione tipica di FCM prevede due componenti principali:

- **Server / ambiente trusted**: backend che invia i messaggi tramite HTTPS o Admin SDK, gestendo targeting, priorità, TTL e tracciamento.

- **Client app**: gestisce i token dei dispositivi, la registrazione e i callback alla ricezione dei messaggi, supportata da Google Play Services (Android) o APNs (iOS).

3.4.4 Tipologie di messaggi e opzioni di consegna

FCM supporta due tipologie principali:

- **Messaggi di notifica**: consegnati e mostrati direttamente dal sistema operativo (titolo, testo, badge, suono).
- **Messaggi di dati**: pacchetti fino a 4 KB, elaborati direttamente dall'applicazione.

Opzioni avanzate di consegna:

- **Collapse key**: sostituzione dei messaggi non ancora recapitati con quelli più recenti.
- **Priorità**: normale (a risparmio energetico, consegna più lenta) o alta (consegna immediata).
- **Time-to-Live (TTL)**: validità del messaggio fino a 28 giorni; se TTL=0 e il device è offline, il messaggio viene scartato.

3.4.5 Targeting e segmentazione

Le modalità di targeting offerte da FCM:

- **Device token-based**: invio a un singolo dispositivo.
- **Messaggio gruppale**: invio a gruppi di dispositivi registrati.
- **Messaggi tematici**: invio a utenti sottoscritti a un argomento (topic).
- **Audience targeting via Analytics**: segmenti dinamici basati su comportamento, demografia o versione dell'app.

3.4.6 Gestione in-app e funzionalità avanzate

Gli sviluppatori possono usare Cloud Functions per notifiche automatiche legate a eventi (es. modifiche a Firestore). Sono supportati test A/B e tracciamento conversioni tramite Firebase Console.

3.4.7 Vantaggi in fase di sviluppo e implementazione

- **Semplice configurazione**: integrazione dell'SDK con poche righe di codice.
- **Minimo codice server**: uso della Console o di Cloud Functions riduce il lavoro backend.
- **Ecosistema integrato**: piena compatibilità con Analytics, A/B Testing, Remote Config e Crashlytics.
- **Scalabilità automatica**: gestisce milioni di messaggi grazie all'infrastruttura Google Cloud.

3.4.8 Casi d'uso tipici

- **Chat e messaggistica**: notifiche push per nuovi messaggi o inviti.
- **Aggiornamenti in tempo reale**: app di news, finanza o sport che richiedono latenza minima.
- **Re-engagement e promozioni**: offerte personalizzate, reminder e messaggi di ritorno.
- **Attivazioni automatiche**: eventi generati da Firestore o altri servizi che innescano notifiche automatiche.

3.4.9 Limiti e considerazioni

Nonostante i vantaggi, FCM presenta alcuni limiti: ritardi occasionali nella consegna broadcast/topic, mancata consegna se i token scadono, e limitazioni per casi ultra real-time (latenze < 100 ms).

3.4.10 Conclusioni

Firebase Cloud Messaging è una soluzione di riferimento per la gestione delle notifiche push e dei messaggi dati. È semplice da integrare, scalabile e gratuito, con opzioni avanzate di targeting. Non è adatto a scenari ultra real-time, ma resta ottimale per chat, news, reminder ed engagement, grazie all'integrazione nativa con l'ecosistema Firebase.

3.5 Visione Generale dell'Architettura

Il sistema sviluppato è stato progettato per coprire l'intero ciclo produttivo del pollame, a partire dalla fase di acquisizione degli animali fino alla macellazione. L'architettura adottata si fonda su principi di **scalabilità**, **modularità** e **resilienza**, elementi fondamentali per garantire affidabilità e capacità di adattamento a carichi di lavoro variabili.

L'infrastruttura è basata su un'architettura a microservizi, orchestrata tramite **Kubernetes**, che consente di distribuire i vari componenti in container indipendenti, facilmente scalabili e gestibili. Kubernetes svolge un ruolo centrale nel garantire deployment automatizzati, bilanciamento del carico e alta disponibilità.

La gestione e il monitoraggio dei processi aziendali sono affidati a **Camunda 8**, una piattaforma open-source pensata per l'esecuzione e l'orchestrazione di workflow BPMN. Camunda funge da cuore logico dell'intero sistema, consentendo di automatizzare, supervisionare e analizzare i processi.

Il sistema si articola nei seguenti componenti principali:

- **Camunda 8**: piattaforma di orchestrazione dei processi che include moduli fondamentali come Zeebe, Operate, Tasklist, Identity, Optimize e Connectors.
- **Zeebe Worker**: motore di logica automatica custom, sviluppato per gestire attività specifiche non coperte dai worker standard di Camunda.
- **BFF (Backend For Frontend)**: ponte tra il frontend e i servizi backend, che semplifica l'interazione con gli utenti e adatta i dati alle varie interfacce.
- **Frontend Web Responsive**: interfaccia utilizzabile da qualsiasi dispositivo, progettata per essere intuitiva e adattabile a diversi schermi.
- **Firebase**: servizio per l'invio di notifiche push, utilizzato per aggiornare gli utenti sugli eventi più rilevanti.
- **Keycloak**: sistema per l'autenticazione e la gestione degli accessi, con supporto ai protocolli standard `OAuth2` e `OpenID Connect`.

Questi componenti lavorano insieme per garantire che il sistema sia sempre affidabile e funzionante, anche in presenza di guasti o picchi di utilizzo.

3.6 Descrizione Dettagliata dei Servizi

3.6.1 Camunda 8

Camunda 8 costituisce il nucleo dell'orchestrazione dei processi e può essere considerata il cervello del sistema. È composta da una serie di container indipendenti, ciascuno con una funzione ben definita:

- **Zeebe Broker**: come descritto alla sezione 3.1.1
- **Operate**: come descritto alla sezione 3.1.1
- **Tasklist**: come descritto alla sezione 3.1.1
- **Optimize**: come descritto alla sezione 3.1.1
- **Connectors**: permettono l'integrazione con sistemi e applicazioni esterne tramite connettori predefiniti. Questa caratteristica semplifica lo scambio di dati e consente l'automazione di processi complessi che coinvolgono più piattaforme.
- **Identity**: controlla gli utenti, i gruppi e i permessi. È integrato con Keycloak per garantire un controllo sicuro e costante degli accessi. In questo modo, è possibile definire autorizzazioni molto specifiche e centralizzare la gestione degli utenti in un unico punto.

3.6.2 Zeebe Worker

Il Zeebe Worker rappresenta un componente fondamentale del sistema di orchestrazione. È un servizio esterno e autonomo che stabilisce una connessione diretta con il broker Zeebe, al fine di eseguire in maniera automatizzata i service task definiti all'interno dei processi BPMN. Tali task incarnano operazioni specifiche e critiche per la logica di business, tra cui:

- **Verifica dei dati**: controllo dell'integrità e della coerenza delle informazioni inserite nel processo.
- **Avvio di sottoprocessi**: consente l'attivazione di flussi secondari legati al processo principale, permettendo la gestione di attività parallele senza complicare il modello principale.
- **Elaborazioni ad hoc**: consentono la creazione di logiche applicative specifiche, adattate alle circostanze aziendali reali.

Il worker può essere sviluppato in vari linguaggi contemporanei, come Java, Go, Kotlin o Node.js, in base alla tecnologia utilizzata dall'azienda e alle capacità del team. Nella nostra situazione, è stato scelto Java. L'uso del client ufficiale per comunicare con Zeebe garantisce un'integrazione coerente, sicura e conforme alle best practice della piattaforma.

La gestione di un database privato è un aspetto distintivo del worker:

- Garantisce un alto livello di sicurezza, poiché l'accesso è riservato al solo worker.
- I dati necessari alla logica di business vengono conservati internamente senza essere esposti ad altri servizi.
- I ruoli e le responsabilità all'interno del sistema sono distinti.

Il worker si occupa della fase di bootstrap del processo oltre all'esecuzione di singole attività. Durante l'inizializzazione, incorpora i dati più aggiornati e coerenti nel contesto Camunda, garantendo l'avvio regolare dei flussi e la corretta tracciabilità delle operazioni.

3.6.3 BFF - Backend For Frontend

Il Backend For Frontend (BFF) funge da intermediario tra il frontend e i vari microservizi interni. Le sue funzioni sono molteplici e cruciali per garantire un accesso sicuro, efficiente e personalizzato alle risorse del sistema. In particolare, il BFF:

- Espone un'API REST progettata su misura per il frontend, permettendo un accesso ottimizzato e performante.
- Recupera e aggrega dati provenienti da Camunda, dal worker Zeebe e da Keycloak.
- Esegue validazioni dei dati prima che una task possa essere marcata come completata, garantendo l'affidabilità delle operazioni.
- Applica controlli di accesso granulari, verificando che solo gli utenti autorizzati possano interagire con specifiche risorse.
- Gestisce le notifiche push, includendo la logica di salvataggio, invio e recupero tramite il proprio database interno.

Il BFF funge anche da **gateway di sicurezza**, verificando ogni richiesta e consentendo l'accesso solo agli utenti con i diritti necessari, bloccando tutti i tentativi non autorizzati.

3.6.4 Frontend Web Responsive

Per migliorare l'accessibilità e l'esperienza utente, non è stata utilizzata la Tasklist standard di Camunda, poiché limitata nella personalizzazione, poco responsiva e difficilmente utilizzabile da dispositivi mobili. Al suo posto, è stato realizzato un frontend custom, progettato per garantire:

- Interfaccia su misura, adattata ai processi aziendali.
- Responsività completa, da desktop a smartphone.
- Funzionalità avanzate, tra cui:
  - Visualizzazione della cronologia delle notifiche.
  - Controlli sui dati per ottenere informazioni dettagliate.
  - Consultazione dei documenti direttamente dall'interfaccia utente.

Tutte le interazioni tra frontend e backend avvengono esclusivamente tramite le API esposte dal BFF, garantendo un accesso sicuro e controllato.

3.6.5 Firebase

Firebase è utilizzato per la gestione delle notifiche push, lavorando in stretta collaborazione con il BFF, che ne orchestra l'intero funzionamento. In particolare:

- Il BFF mantiene un database interno con tutte le notifiche inviate e ricevute.
- I token di registrazione Firebase vengono associati ai browser attivi degli utenti.
- Quando si verifica un evento rilevante (es. creazione di una nuova task o completamento di uno step), il BFF:
  1. salva la notifica completa nel proprio database,
  2. invia una versione ridotta tramite Firebase,
  3. espone un endpoint dedicato al recupero dei dettagli completi dal frontend.

Questo approccio garantisce un sistema di notifiche affidabile, scalabile e ben integrato nel flusso complessivo dell'applicazione.

3.6.6 Keycloak

Keycloak gestisce l'autenticazione e l'autorizzazione degli utenti e funge da nucleo del sistema di gestione delle identità. Le sue principali responsabilità sono:

- **Registrazione degli utenti tramite browser**: gestione completa dell'accesso con credenziali proprie, riducendo la possibilità di accessi non autorizzati.
- **Assegnazione a gruppi e ruoli**: permette di suddividere i permessi in base alle responsabilità, distinguendo tra operatori, manager e amministratori.
- **Integrazione con Identity di Camunda**: garantisce una gestione centralizzata delle identità, unificando il controllo degli accessi tra orchestratore e servizi applicativi.
- **Emissione di token JWT per l'accesso sicuro alle API**: consente di validare le richieste verso le API, garantendo che solo utenti autenticati e autorizzati possano accedere alle risorse.

Le regole di accesso impostate da Keycloak consentono di gestire i permessi in modo dettagliato, garantendo privilegi distinti per ogni ruolo aziendale.

3.7 Pattern di Comunicazione

3.7.1 Pattern interni al cluster

I servizi nel cluster Kubernetes comunicano attraverso il DNS interno, che garantisce un instradamento efficiente e veloce. I principali pattern di comunicazione interni includono:

- **Zeebe Worker → Zeebe Broker (gRPC)**: i worker si collegano ai broker per l'esecuzione dei service task tramite protocollo gRPC, ottimizzato per alte prestazioni.
- **BFF → Camunda (gRPC)**: il Backend For Frontend comunica con Camunda utilizzando gRPC, necessario per recuperare e aggiornare lo stato dei processi.
- **BFF → Worker (REST)**: il BFF interagisce con i worker attraverso la comunicazione REST.
- **Frontend → BFF (REST)**: il frontend si collega esclusivamente al BFF tramite protocollo REST, garantendo sicurezza e affidabilità nella comunicazione.

- **BFF → Firebase (HTTPS)**: il BFF utilizza connessioni HTTPS per inviare notifiche push tramite Firebase.

3.7.2 Pattern esterni al cluster

L'unico componente accessibile dall'esterno è il BFF, che rappresenta il punto di ingresso autenticato per l'utente finale. Gli ingress controller, come ad esempio NGINX Ingress, si occupano di instradare il traffico in arrivo, garantendo al tempo stesso che la rete interna resti isolata da quella proveniente dall'esterno.

3.8 Gestione della Sicurezza

L'infrastruttura è progettata con un approccio a più livelli di protezione, garantendo un bilanciamento tra sicurezza e flessibilità operativa. I principali meccanismi implementati sono:

- **Autenticazione**: gestita centralmente da Keycloak, con emissione e validazione di token JWT.
- **Autorizzazione**: effettuata dal BFF sulla base dei ruoli e dei gruppi assegnati agli utenti.
- **Accesso ai dati sensibili**: i database vengono separati tra i diversi componenti (ad esempio worker e BFF), in modo che soltanto i servizi autorizzati possano consultare le informazioni che gli competono.
- **Validazione dei dati**: i controlli sui dati vengono eseguiti sia dal lato client (frontend) sia dal lato server (BFF), così da evitare errori e mantenere coerenza tra le due parti.
- **Crittografia**: tutte le comunicazioni, interne ed esterne, avvengono tramite protocollo HTTPS; inoltre, i dati sensibili presenti nei database vengono memorizzati in forma crittografata.

3.9 Notifiche Push

Per ridurre il carico e mantenere sempre la tracciabilità delle comunicazioni, il sistema di notifiche push è stato progettato per essere affidabile, scalabile e asincrono. Il funzionamento può essere riassunto come segue:

1. Firebase assegna un token specifico a ogni browser, che identifica in modo sicuro il dispositivo.

2. Il BFF conserva questo token e lo associa all'utente corrispondente.

3. Quando deve essere inviata una notifica, il BFF registra tutti i dettagli nel database e recapita al client una versione semplificata tramite Firebase.

4. Dopo aver ricevuto la notifica leggera, il client ricontatta il BFF per ottenere i contenuti completi.

Questo approccio consente di:

- Ridurre il carico su Firebase.
- Garantire tracciabilità e affidabilità delle notifiche inviate e ricevute.

3.10 Motivazioni Progettuali

3.10.1 Worker dedicato

La scelta di utilizzare un worker dedicato per la logica automatica si traduce in numerosi vantaggi architetturali:

- Consente di mantenere isolata la business logic dal resto del sistema.
- Semplifica e velocizza i test.
- Permette di scalare i worker indipendentemente dall'interfaccia utente o dall'orchestratore Camunda.

3.10.2 Back-End For Frontend e funzionalità dell'architettura

L'implementazione di Back-End For Frontend (BFF) ha reso possibile concentrare alcune funzioni essenziali dell'architettura, rendendola più robusta e facile da gestire. Specificamente, il BFF si occupa di:

- **Validazione dei dati**: applica regole comuni di controllo prima di inviare i dati ai servizi interni. In questo modo non vengono duplicati controlli sui vari frontend, riducendo la probabilità di errori o incoerenze.
- **Sicurezza dei flussi**: facilita l'interazione tra frontend e servizi, controllando autenticazione e autorizzazione per assicurare che solo utenti autorizzati possano accedere a funzioni e dati sensibili.

- **Aggregazione di dati da più fonti**: raccoglie e unisce informazioni da diverse origini in una risposta coerente, evitando che i frontend interroghino direttamente più servizi e database.
- **Integrazione con sistemi esterni**: funge da punto di contatto per servizi esterni come notifiche push, orchestratore Camunda e database collegati ai worker, semplificando il frontend e riducendo la frammentazione.

3.10.3 Frontend Custom

L'interfaccia utente custom è stata progettata con particolare attenzione all'esperienza d'uso e ai requisiti specifici di business. Essa garantisce:

- **Responsività mobile**: il design si adatta automaticamente a dispositivi diversi, dai desktop ai tablet fino agli smartphone, assicurando accessibilità e continuità d'uso.
- **Personalizzazione dell'interfaccia per task specifici**: le schermate sono modellate sulle esigenze dell'utente e sul contesto operativo, mostrando solo gli elementi necessari e guidando l'operatore passo dopo passo.
- **Branding e UX su misura**: l'identità aziendale viene adattata al design grafico e alle modalità di interazione, garantendo coerenza con il brand e rendendo l'esperienza più chiara e gradevole.

3.11 Esempio di Flusso

Un ciclo tipico di utilizzo della piattaforma segue i passaggi riportati di seguito:

1. L'utente (operatore) accede all'applicazione tramite il frontend e avvia un nuovo processo.
2. Il frontend invia una richiesta al BFF, che inizializza il processo sul motore Camunda.
3. Camunda assegna un task automatico a un worker dedicato, incaricato di elaborare i dati iniziali ed eseguire eventuali calcoli preliminari.
4. Il worker salva i risultati nel proprio database e restituisce le informazioni aggiornate al BFF.
5. Il BFF invia una notifica push all'utente, segnalando la disponibilità di un nuovo task manuale da completare.

6. L'utente accede nuovamente all'interfaccia, compila un modulo personalizzato sul frontend e invia i dati richiesti.

7. Il BFF valida accuratamente i dati ricevuti, quindi li inoltra a Camunda per consentire l'avanzamento del processo allo step successivo.

3.12 Conclusione

L'architettura descritta garantisce al tempo stesso flessibilità, sicurezza e scalabilità. Ogni componente, dal frontend custom al BFF, fino ai worker specializzati, è progettato con responsabilità ben definite, evitando sovrapposizioni e semplificando la manutenzione.

L'uso di Camunda 8 come orchestratore, integrato con un BFF intelligente e con worker dedicati, permette di gestire in modo moderno ed efficiente workflow mission-critical in ambienti enterprise. Questo approccio modulare consente all'intero ecosistema di adattarsi facilmente all'aumentare della complessità e del volume dei processi da gestire.

3.13 Approfondimenti Tecnici (Aggiunti)

3.13.1 Esempi di Deploy su Kubernetes

Di seguito è riportato un esempio di manifest Kubernetes per il deployment di un microservizio Spring Boot e di un Zeebe broker. Questi esempi sono pensati come modello di partenza e vanno adattati all'ambiente di produzione.

```
# Deployment Spring Boot (bff-deployment.yaml)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bff-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: bff
  template:
    metadata:
      labels:
        app: bff
    spec:
```

```
      containers:
      - name: bff
        image: registry.example.com/bff:latest
        ports:
        - containerPort: 8080
        resources:
          requests:
            cpu: "250m"
            memory: "256Mi"
          limits:
            cpu: "1"
            memory: "1Gi"
        env:
        - name: SPRING_PROFILES_ACTIVE
          value: "prod"
```

3.13.2 Configurazione di Zeebe (esempio semplificato)

```
# StatefulSet per Zeebe broker (semplificato)
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: zeebe-broker
spec:
  serviceName: "zeebe"
  replicas: 3
  selector:
    matchLabels:
      app: zeebe
  template:
    metadata:
      labels:
        app: zeebe
    spec:
      containers:
      - name: zeebe
        image: camunda/zeebe:8.7.0
        ports:
        - containerPort: 26500
```

3.13.3 Esempio di Zeebe Worker in Kotlin

Un semplice worker in Kotlin che ascolta un task type e completa un job.

```
import io.camunda.zeebe.client.ZeebeClient

fun main() {
  val client = ZeebeClient.newClientBuilder()
      .brokerContactPoint("zeebe-broker:26500")
      .usePlaintext()
      .build()

  client.newWorker()
      .jobType("send-notification")
      .handler { client, job ->
          val variables = job.variablesAsMap()
          // logica di invio notifica...
          client.newCompleteCommand(job.key).send().join()
      }
      .open()
}
```

3.13.4 Esempio di Controller Spring Boot (Kotlin) per il BFF

```
@RestController
@RequestMapping("/api/v1/tasks")
class TaskController(val taskService: TaskService) {

  @GetMapping("/{id}")
  fun getTask(@PathVariable id: String): ResponseEntity<TaskDto> {
    val task = taskService.getTask(id)
    return ResponseEntity.ok(task)
  }

  @PostMapping("/{id}/complete")
  fun completeTask(@PathVariable id: String): ResponseEntity<Void> {
    taskService.completeTask(id)
    return ResponseEntity.noContent().build()
  }
}
```

3.13.5 Configurazione OAuth2/OIDC (Spring Security)

Esempio di configurazione minimale per integrare Keycloak (o altro IdP) con Spring Security usando JWT e OIDC.

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://auth.example.com/realms/yourrealm
```

3.13.6 Esempi di API del BFF verso Camunda 8 (Tasklist/Operate)

```
# esempio: completare un task tramite TasklistClient (pseudo)
POST /bff/tasks/{taskId}/complete
Authorization: Bearer <token>
Body: { "variables": { "approved": true } }
```

3.13.7 Tabelle di confronto e metriche di dimensionamento

- **Replica count**: definire in base al carico atteso, alla latenza e ai requisiti RTO/RPO.
- **Requests/Limiti**: assegnare risorse conservative in fase di test, aumentandole in produzione in base al monitoraggio.
- **Monitoraggio**: utilizzare Prometheus + Grafana per metriche e alerting. Configurare soglie per CPU, memoria e latenza delle API.

3.13.8 Esempio di Schema DB (DDL semplificato)

```
CREATE TABLE tasks (
  id UUID PRIMARY KEY,
  process_instance_id VARCHAR(64),
  assignee VARCHAR(128),
  payload JSONB,
  created_at TIMESTAMP WITH TIME ZONE DEFAULT now()
);
```

3.13.9 Strategie di Test e CI/CD

- Unit tests: JUnit + MockK per Kotlin.
- Integration tests: Testcontainers per simulare dipendenze come DB e Zeebe.
- E2E: pipeline che esegue scenari BPMN completi in ambiente di staging.

3.13.10 Esempi di Query Operative (Elasticsearch)

```
# ricerca per istanza processo e stato
GET /process-instance/_search
{
  "query": { "term": { "processInstanceId": "1234-uuid" } }
}
```

3.13.11 Considerazioni sulla Sicurezza Operativa

- Isolare le reti interne (Kubernetes namespaces, network policies).
- Abilitare TLS tra i componenti (gateway, broker, client).
- Rotazione automatica delle chiavi e gestione dei segreti (HashiCorp Vault).

3.13.12 Appendice: Checklist per il rilascio in produzione

1. Validare i backup del database.
2. Test di failover del cluster Zeebe/Kubernetes.
3. Validare le regole di sicurezza e i permessi Keycloak.
4. Stress test con carichi realistici.

Capitolo 4

Backend-for-Frontend

4.1 Introduzione

Il presente capitolo è dedicato all'analisi approfondita del *Backend-for-Frontend* (BFF), un componente cardine, sviluppato nell'ambito di un sistema complesso, che si occupa della gestione integrale del ciclo produttivo del pollame. L'intero processo copre tutte le fasi operative, a partire dall'acquisto e dalla registrazione iniziale degli animali, fino a giungere alla loro macellazione, comprendendo dunque un insieme articolato di attività eterogenee e fortemente interconnesse.

Il BFF è una parte importante dell'architettura del sistema. È un punto di coordinamento e collega *frontend* e *backend*. Si collega a numerosi sottosistemi essenziali e interagisce con l'interfaccia utente in modo semplice e coerente. Questi includono il motore di orchestrazione Camunda 8, che crea ed esegue i workflow; il *worker* Zeebe, che si occupa dell'automazione dei compiti; i sistemi di autorizzazione e autenticazione basati su Keycloak, che forniscono protezione; e i servizi di notifica, come Firebase, che consentono agli operatori di comunicare in tempo reale.

Il compito del BFF va dunque ben oltre la semplice intermediazione. Esso incapsula logiche complesse che riguardano: la gestione della sicurezza dei flussi e la protezione delle informazioni sensibili; l'aggregazione e la trasformazione di dati provenienti da fonti eterogenee; la validazione delle richieste e dei payload in transito; nonché l'ottimizzazione della comunicazione tra le diverse componenti, così da ridurre la latenza e migliorare le prestazioni complessive. In questo modo il BFF diventa un punto di accesso unico e controllato, capace di garantire sia efficienza che affidabilità.

Lo sviluppo di tale componente non è stato immediato né lineare. Al contrario, ha richiesto mesi di progettazione, implementazione e numerose iterazioni di revisione. Il BFF, infatti, non rappresenta un modulo statico, ma un'entità in continua evoluzione. Le modifiche ai requisiti funzionali, l'introduzione di nuove tecnologie

nella piattaforma Camunda e la necessità di integrare strumenti emergenti hanno spinto a ripensare più volte le sue logiche interne e le sue interfacce. Questo percorso ha portato a trasformazioni profonde, che hanno progressivamente arricchito il sistema di funzionalità e ne hanno migliorato la resilienza.

All'interno di questo capitolo verranno approfondite le motivazioni che hanno guidato le principali scelte architetturali, evidenziando come esse siano state influenzate da esigenze tecniche, organizzative e di scalabilità. Si analizzeranno i problemi incontrati nel corso dello sviluppo, mettendo in luce le criticità emerse e le strategie adottate per affrontarle. Saranno descritti nel dettaglio gli strumenti introdotti a supporto del ciclo di vita del software, dalle librerie di integrazione alle soluzioni di monitoraggio.

Ampio spazio sarà riservato sia agli aspetti prettamente tecnici, come la gestione delle API, il passaggio dall'approccio REST tradizionale ai client `TaskListClient` e `OperateClient`, e le attività di refactoring del codice , sia a quelli di natura metodologica, comprendenti le strategie di testing automatizzato, il miglioramento continuo attraverso feedback iterativi e la collaborazione con la comunità open-source di Camunda. L'insieme di queste analisi permetterà di restituire una visione completa e articolata del ruolo del BFF, sottolineandone il valore aggiunto per l'intera architettura.

4.2 Architettura Iniziale

4.2.1 Separazione dei Servizi

Nel corso della fase iniziale di ideazione e progettazione del sistema, è stata presa la decisione di utilizzare un'architettura basata sulla divisione delle responsabilità tra due microservizi separati. In modo da garantire maggiore flessibilità, chiarezza e scalabilità indipendente, l'obiettivo principale era dividere i compiti più importanti, mantenendo le logiche di comunicazione dalle logiche di gestione delle notifiche. I due microservizi identificati sono stati:

- **Proxy Service**: è stato progettato per gestire tutte le richieste dal *frontend*. Il suo lavoro include funzioni di filtro e controllo preliminare, oltre all'inoltro verso le API REST di Camunda. In questa fase, il Proxy Service verifica la correttezza delle richieste, applica le regole di sicurezza per l'accesso alle risorse e centralizza la logica di validazione. Questo funge da primo livello di protezione e da strato di astrazione rispetto ai servizi sottostanti.

- **Notification Service**: dedicato in maniera specifica alla gestione completa delle notifiche generate dal sistema. Questo servizio si occupava di tre attività principali: la creazione delle notifiche in risposta ad eventi di business; la loro persistenza all'interno di un datastore dedicato, così da garantire la

tracciabilità nel tempo; e infine l'invio agli utenti finali tramite la piattaforma Firebase, che consentiva di raggiungere rapidamente sia applicazioni mobili sia interfacce web.

L'adozione di questa architettura modulare era stata motivata dall'ipotesi di dover gestire scenari di carico molto eterogenei. Si era previsto, ad esempio, che il *Notification Service* potesse essere sottoposto a picchi significativi di eventi da processare, mentre il *Proxy Service* avrebbe mantenuto un flusso più stabile e prevedibile di richieste. In tale ottica, la possibilità di scalare in maniera indipendente i due componenti rappresentava un potenziale vantaggio, riducendo il rischio che un sovraccarico in una specifica area del sistema potesse compromettere le altre funzionalità. Un simile approccio è piuttosto diffuso nelle architetture a microservizi, dove la differenziazione dei carichi di lavoro costituisce una delle ragioni principali per giustificare la separazione dei servizi.

Tuttavia, un'analisi più approfondita del dominio applicativo e dei flussi operativi reali ha mostrato un quadro differente. È emerso infatti che il numero di notifiche prodotte non costituiva un parametro indipendente, bensì risultava strettamente proporzionale al numero di operazioni eseguite sui processi. In altre parole, il carico di lavoro dei due microservizi non era eterogeneo, ma sostanzialmente correlato e non si osservavano picchi tali da giustificare l'onere di mantenere due unità distinte.

4.2.2 Criticità della Modularità Iniziale

La scelta iniziale di mantenere due microservizi separati, pur teoricamente corretta e motivata dalla volontà di garantire flessibilità, ha introdotto una serie di complessità operative che si sono rese evidenti fin dalle prime fasi di esercizio:

- **Maggiore complessità di deploy**: l'utilizzo di Kubernetes per la gestione dell'infrastruttura comportava due pipeline di rilascio indipendenti. Ogni modifica, anche marginale, a uno dei servizi richiedeva un ciclo di build, test e distribuzione autonomo, aumentando i tempi complessivi di rilascio e moltiplicando le possibilità di errore umano.

- **Superficie di attacco ampliata**: dal punto di vista della sicurezza, mantenere due endpoint distinti esposti verso l'esterno significava aumentare le potenziali vulnerabilità. Ogni servizio costituiva infatti un ulteriore punto di ingresso che richiedeva protezione, monitoraggio e aggiornamenti costanti.

- **Necessità di sincronizzazione e coordinamento**: avere due basi di codice separate comportava un lavoro aggiuntivo di coordinamento tra team e sviluppatori. Gestire dipendenze condivise, versioni compatibili e flussi di comunicazione interservizio introduceva un overhead gestionale significativo, rallentando lo sviluppo e complicando le attività di manutenzione.

Si è deciso di abbandonare la modularità iniziale per un approccio più semplice ed efficace, tenendo conto di queste criticità e dopo un primo periodo di test in ambiente reale.

4.3 Interazione Iniziale con Camunda

4.3.1 Traduzione 1:1 delle API

Nella fase iniziale dell'implementazione unificata, il *backend-for-frontend* di Textit (BFF) si collegava a Camunda solo utilizzando le API REST fornite dalla piattaforma. Il metodo è stato scelto per essere semplice, poiché ogni endpoint esposto dal BFF aveva un corrispondente diretto in Camunda. Pertanto, il lavoro principale del BFF era quello di fungere da livello intermedio di inoltro, implementando controlli di sicurezza e validazioni iniziali, senza alterare i dati in modo significativo.

Sebbene volutamente poco costoso, questo metodo ha fornito alcuni vantaggi immediati:

- **Stabilire un canale di comunicazione funzionante**: il team ha potuto avviare rapidamente uno scambio operativo tra il frontend e Camunda grazie alla traduzione 1:1 delle API. Ciò ha ridotto i tempi di integrazione iniziali.
- **Concentrare gli sforzi sulla sicurezza**: la mancanza di logiche di trasformazione dei dati ha permesso di concentrarsi sulla definizione e sull'attuazione delle norme di accesso e protezione delle risorse, che sono considerate più importanti delle ottimizzazioni prestazionali.
- **Fornire stabilità al frontend**: l'introduzione di un livello intermedio garantiva al frontend un'interfaccia uniforme e stabile, anche nel caso in cui le API sottostanti di Camunda subissero modifiche evolutive durante lo sviluppo.

In sintesi, questo modello 1:1 rappresentava una scelta consapevolmente pragmatica, orientata più alla rapidità di messa in esercizio e alla solidità del sistema che non all'efficienza assoluta.

4.3.2 Pseudo-codice del Flusso Iniziale

Il frammento seguente illustra la logica di inoltro tipica di questa fase, prendendo come esempio una chiamata `POST` per la ricerca delle attività utente.

```
1 fun getTasks(username: String, group: List<String>): List<TaskDTO>
      {
2     val requestBody = {"candidateUser": username}
3     val tasks = apiCallsHelper.sendPostRequest(
```

```
        "$camundaUrl/v1/tasks/search",
        requestBody,
        Array<Task>::class.java
    )
    return tasks.map(it.toDTO())
}
```

Listing 4.1: Pseudo-codice della logica iniziale di inoltro delle richieste, esempio di una POST

Il funzionamento può essere descritto come segue:

- La funzione `getTasks` riceve come input il nome utente e la lista dei gruppi di appartenenza.
- Viene costruito un oggetto JSON di richiesta (`requestBody`) che incapsula i parametri necessari, in questo caso l'utente candidato.
- L'oggetto viene inviato tramite l'helper `apiCallsHelper` all'endpoint REST `/v1/tasks/search` di Camunda, utilizzando una chiamata `POST`.
- La risposta, tipizzata come array di `Task`, viene deserializzata e trasformata in una lista di `DTO` (Data Transfer Object), più adatti ad essere consumati dal frontend.
- Prima dell'invio della richiesta, è stata applicata la logica di sicurezza, che non è stata mostrata esplicitamente nello snippet, per garantire che solo utenti autorizzati potessero accedere a determinate risorse.

Lo schema sottolinea la natura di semplice inoltro del BFF nelle sue prime fasi, evidenziando come l'architettura fosse inizialmente dedicata alla velocità di sviluppo e alla solidità dei controlli di accesso, piuttosto che all'ottimizzazione o alla trasformazione dei dati.

4.4 Strumenti di Supporto alle API

4.4.1 Classe di Gestione delle Chiamate

Per semplificare l'interazione con le API di Camunda e ridurre la duplicazione di codice, è stata sviluppata una classe di utilità centralizzata. La classe ha l'obiettivo di centralizzare la gestione delle chiamate HTTP, uniformare la gestione degli errori e fornire un punto unico di manutenzione per le comunicazioni con il motore.

```
fun <T> sendGetRequest(
    url: String,
    mappedType: Class<T>,
```

```
4       authType: AuthType = AuthType.SERVICE_ACCOUNT
5   ): T {
6       val request = HttpRequest.newBuilder()
7           .uri(URI.create(url))
8           .header("Content-Type", "application/json")
9           .header("Authorization", "Bearer ${getAuthToken(authType)}
        ")
10          .header("Accept", "application/json")
11          .timeout(Duration.ofMillis(requestTimeoutMs))
12          .GET()
13          .build()
14      return executeRequestHelper(request, mappedType)
15  }
```

Listing 4.2: Metodo generico per richieste GET verso Camunda

Questa classe consente di:

- Ridurre la complessità per gli sviluppatori, permettendo loro di concentrarsi sulla logica di business principale.
- Gestire in modo uniforme i token di autenticazione, sia per account di servizio sia per utenti finali.
- Facilitare la manutenzione futura, centralizzando la logica comune per tutte le chiamate GET alle API di Camunda.

Per riassumere, è stato possibile creare un BFF robusto, sicuro e facilmente estendibile attraverso l'implementazione di controlli di sicurezza sofisticati e strumenti di supporto per le API.

4.5 Controlli di Sicurezza

4.5.1 Necessità di Validazione Avanzata

La piattaforma Camunda presenta un limite intrinseco: le sue impostazioni predefinite sono piuttosto permissive. Le API standard non impediscono azioni non autorizzate; ad esempio, un utente potrebbe completare un task non assegnato a lui. In un ambiente produttivo, questa condizione rappresenta un rischio significativo poiché le operazioni eseguite potrebbero avere impatti economici o gestionali diretti. È quindi necessario introdurre controlli di sicurezza più rigorosi per garantire l'integrità dei processi.

4.5.2 Implementazione dei Controlli

Il BFF (*Backend for Frontend*) ha creato una serie di controlli di sicurezza sofisticati per ridurre questi pericoli. Questi controlli garantiscono che solo gli utenti autorizzati e autenticati possano fare determinate cose. In particolare, sono stati messi in atto:

- **La validazione JWT**: richiede che ogni richiesta contenga un *JSON Web Token* (JWT) firmato digitalmente tramite Keycloak. La validità del token garantisce che l'utente mittente sia autentico e impedisce l'accesso non autorizzato.
- **Controllo dei ruoli e dei gruppi**: il sistema verifica che l'utente appartenga a gruppi autorizzati per svolgere le azioni richieste. Solo i membri di gruppi specifici sono autorizzati a svolgere operazioni specifiche, fornendo così un livello aggiuntivo di protezione.
- **Verifica dell'assegnazione della task**: il BFF verifica che la task sia effettivamente assegnata all'utente autenticato prima di consentire l'esecuzione. Questo evita che il personale non autorizzato manipoli i compiti e protegge il flusso di lavoro.

4.6 Migrazione a Camunda 8.7

4.6.1 Introduzione di TaskListClient e OperateClient

Molte delle API che erano disponibili prima dell'aggiornamento a Camunda 8.7 sono state deprecate o eliminate, costringendo le applicazioni a utilizzare i nuovi client Java ufficiali **TaskListClient** e **OperateClient**. Questi nuovi client facilitano l'interazione con compiti, processi e variabili e rendono il lavoro degli sviluppatori più chiaro e affidabile.

- **TaskListClient**: gestisce i task di Camunda, permettendo operazioni come completamento, assegnazione e recupero di variabili.
- L'API di Operate consente di interrogare processi, compiti e incidenti.
- I problemi con le API: le API REST GraphQL e vecchie di Tasklist e Operate non sono più supportate. Non è consigliato utilizzarle e saranno completamente eliminate nelle versioni future.
- Per garantire compatibilità e aggiornamenti futuri, è necessario sostituire i metodi legacy con i client ufficiali o gli endpoint REST di orchestrazione.

4.6.2 Esempio di Completamento Task con TaskListClient

Il codice sotto riportato mostra come il BFF completa un task usando **TaskListClient** e includendo logging e gestione delle variabili :

```
1
2
3  fun completePianificazionePollastre(
4          id: String, myUsername: String, pianificazione:
       PianificazionePollastre
5      ) {
6      //controlla se la task è effettivamente tua
7          val task = taskListClient.getTask(id, false) ?: throw
       TaskNotFound("Task with ID $id not found")
8          if(task.assignee != myUsername) throw NotAssignedToUser("
       Task not assigned to $myUsername or your groups")
9      //recupera o aggiona le variabili per la validazione dei campi
10         val cachedAllevamentiMaturazione = getCachedAllevamenti("
       pollastre")
11         val cachedAllevamentiRiproduzione = getCachedAllevamenti("
       ovodeposizione")
12     // esegue una funzione di validazione
13         checkValidPianificazione(
14             pianificazione.pianificazione,
       cachedAllevamentiMaturazione, cachedAllevamentiRiproduzione
15         )
16     //chiama la funzione che si occupa di effettuare la complete
17         taskServiceUtil.callTaskCompleteHelper(id, myUsername,
       JSONObject(pianificazione))
18     }
```

Listing 4.3: Completamento di un task con callTaskCompleteHelper

```
1  fun callTaskCompleteHelper(id: String, myUsername: String,
       variable: JSONObject) {
2      val formData: Map<String, Any> = run {
3          // Flatten JSON delle variabili da inviare
4          val flatConvertedData: Map<String, Any> = JsonUtils().
       flattenMap(variable.toMap())
5
6          // Recupera le variabili correnti del task e le
       appiattisce
7          val flatVariable = JsonUtils().flattenMap(
8              taskListClient.getTask(id, true).variables.associate {
       it.name to it.value }
9          ).toMutableMap()
10
11         // Sovrascrive o aggiunge nuove variabili
12         flatConvertedData.forEach { (key, value) -> flatVariable[
       key] = value }
```

```
13
14          // Ripristina la struttura originale delle variabili
15          JsonUtils().unflattenMap(flatVariable)
16      }
17
18      try {
19          // Completa il task con TaskListClient
20          taskListClient.completeTask(id, formData)
21      } catch (e: Exception) {
22          throw TaskInternalServerError("Error completing task with
    ID $id: ${e.message}")
23      }
24  }
```

Listing 4.4: Completamento di un task con callTaskCompleteHelper

In questo snippet:

- Si recupera la task e si controlla che sia assegnata all'utente che richiede il completamento
- Si recuperano/aggiornano le variabili in cache
- Si validano i dati della form per la complete della task
- Si recuperano le variabili attuali del task tramite `taskListClient.getTask`.
- I dati della form vengono appiattiti e uniti a quelli esistenti.
- Si completa il task con `taskListClient.completeTask`, passando la mappa finale delle variabili.
- Viene gestito il logging dettagliato e le eventuali eccezioni.

4.6.3 Riepilogo

La migrazione a Camunda 8.7 obbliga all'uso di **TaskListClient** e **OperateClient**, in quanto le vecchie API REST/GraphQL sono state disattivate. L'uso di helper come `callTaskCompleteHelper` garantisce:

- Gestione sicura e tipizzata delle variabili.
- Logging e tracciabilità delle operazioni.
- Compatibilità con le versioni future di Camunda.

4.7 Testing e Qualità del Codice

4.7.1 Obiettivi del Testing

Il test è stato progettato per garantire che il `TaskController` funzioni in modo affidabile e coerente. È possibile sintetizzare così gli obiettivi principali:

- **Verifica delle interazioni con Camunda:** Questi test verificano che le chiamate al `TaskListClient` siano corrette, forniscono risposte coerenti e seguono i contratti delle API ufficiali.
- **Correttezza del mapping dei dati:** Garantisce una corrispondenza precisa tra il dominio di Camunda e l'applicazione trasformando correttamente i dati ricevuti dalle API esterne nei DTO interni.
- **Gestione degli errori e della sicurezza:** I test verificano che il sistema usi i codici HTTP corretti e i messaggi chiari, utilizzando scenari di eccezione come accessi non autorizzati o compiti non assegnati.
- **Simulazione di scenari realistici:** Il mocking delle dipendenze consente di simulare flussi operativi simili a quelli del mondo reale, mantenendo al contempo il `TaskController` lontano da sistemi complessi.

4.7.2 Struttura dei Test

L'implementazione dei test segue una metodologia consolidata nell'ecosistema Java:

- viene utilizzato `JUnit 5` come framework di base per la scrittura dei test;
- `Mockito` consente di creare mock e stub delle dipendenze, in particolare per simulare le risposte del `TaskListClient`;
- `MockMvc`, strumento di Spring Test, permette di simulare chiamate HTTP verso gli endpoint REST del controller, senza avviare un server reale;
- ogni test configura un contesto di sicurezza fittizio con utente autenticato e relativi ruoli, in modo da riflettere fedelmente i vincoli di autorizzazione presenti in produzione.

4.7.3 Esempi di Mocking delle Risposte

Un esempio tipico è rappresentato dal recupero delle task assegnate a un utente autenticato. Il `TaskListClient` viene configurato per restituire una lista predefinita di task, che vengono poi validate tramite asserzioni JSON sugli endpoint REST esposti dal sistema:

```
@Test
fun `test getAllTask returns list of tasks when assigned is true`() {
    val authentication = TestingAuthenticationToken("testuser", "password",
        listOf(SimpleGrantedAuthority("ROLE_USER")))
    SecurityContextHolder.getContext().authentication = authentication

    val tasks = TaskList().apply {
        items = listOf(Task().apply {
            id = "task1"
            name = "Task 1"
            creationDate = "2023-01-01T10:00:00Z"
            assignee = "testuser"
            taskState = TaskState.CREATED
        })
    }

    `when`(taskListClient.getAssigneeTasks("testuser", TaskState.CREATED, 50))
        .thenReturn(tasks)

    mockMvc.perform(get("/api/tasklist/getTasks")
        .param("assigned", "true").with(csrf())
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk)
        .andExpect(jsonPath("$[0].id").value("task1"))
        .andExpect(jsonPath("$[0].nome").value("Task 1"))
}
```

Listing 4.5: Test del recupero task assegnate

In questo modo si verifica che:

- i dati prodotti dal mock del `TaskListClient` vengano esposti correttamente;
- l'API REST rispetti i contratti attesi sia in termini di formato sia di contenuto.

4.7.4 Gestione degli Errori

Una parte cruciale del testing riguarda la verifica della corretta gestione delle condizioni di errore. Ad esempio, il caso in cui un utente tenti di assegnarsi una task non autorizzata deve produrre una risposta `403 Forbidden` con un messaggio esplicativo:

```
@Test
```

```
2  fun `test assignToMe throws Forbidden when user is not authorized
       `() {
3          // Set up a security context
4      val authentication = TestingAuthenticationToken("testuser", "
       password",
5          listOf(SimpleGrantedAuthority("ROLE_USER")))
6      SecurityContextHolder.getContext().authentication =
       authentication
7
8      val task = Task().apply {
9          id = "task1"
10         assignee = "others"
11         candidateUsers = listOf("others")
12     }
13
14     doReturn(task).`when`(taskListClient).getTask("task1", true)
15
16     mockMvc.perform(patch("/api/tasklist/{id}/assignToMe", "task1"
       )
17         .with(csrf())
18         .contentType(MediaType.APPLICATION_JSON))
19         .andExpect(status().isForbidden)
20         .andExpect(content().string("Task not assigned to ${
       authentication.name} or your groups"))
21 }
```

Listing 4.6: Test della gestione Forbidden

Questo garantisce che le restrizioni di sicurezza siano applicate correttamente e che l'utente riceva feedback chiari in caso di operazioni non autorizzate.

4.7.5 Copertura e Manutenibilità

La varietà di casi testati comprende:

- Assegnazione e rimozione assegnazione delle task;
- Ricerca con criteri diversi (assegnate, candidate, per gruppi);
- Recupero dei form associati a una task;
- Gestione delle variabili di processo.

L'uso estensivo del mocking ha molti vantaggi:

- Accelera i cicli di sviluppo riducendo la dipendenza da un'istanza reale di Camunda;

- Rende più semplice la gestione del codice di test e rende più facile apportare modifiche;
- I test stessi descrivono e verificano in modo eseguibile i principali flussi del sistema, creando una sorta di "documentazione vivente".

4.8 Collaborazione Open-Source

Durante le fasi di sviluppo del sistema è emerso un problema critico di *mapping*, che causava la perdita di dati e comprometteva la correttezza complessiva dei risultati. Data la natura della libreria coinvolta — un componente open-source ampiamente utilizzato anche in altri progetti, si è deciso di non limitarsi a una correzione interna, ma di contribuire direttamente al miglioramento della libreria ufficiale. Questa scelta ha portato benefici sia al progetto in corso, che ha potuto risolvere rapidamente la criticità, sia all'intera comunità di sviluppatori che utilizza la stessa libreria.

Il processo di collaborazione con la comunità open-source ha seguito le pratiche consolidate di gestione e condivisione del codice:

- **Apertura di un'*issue* pubblica**: il primo passo è stato documentare il problema in maniera dettagliata sul repository ufficiale. Nell'*issue* sono stati descritti i sintomi del bug, forniti scenari riproducibili e spiegato l'impatto che l'errore aveva sui flussi applicativi. Questo ha permesso ad altri utenti e ai manutentori di comprendere la gravità del problema e di seguire la discussione pubblica.
- **Creazione di una *fork***: successivamente è stata generata una copia (*fork*) del repository principale. Tale operazione ha consentito di intervenire direttamente sul codice sorgente senza modificare immediatamente il progetto originale, mantenendo così un ambiente isolato per sperimentare e applicare la correzione. In questa fase è stata implementata la soluzione e ne è stata verificata l'efficacia attraverso test mirati.
- **Invio di una *pull request***: una volta validata la soluzione, è stata aperta una *pull request* verso il repository ufficiale. Questo passaggio ha reso disponibile la modifica all'intera comunità, permettendo ai maintainer di esaminare il codice, commentarlo e integrarlo nel ramo principale. Il meccanismo della pull request garantisce trasparenza e revisione paritaria, elementi centrali nello sviluppo collaborativo.

L'impatto di questo intervento è stato estremamente positivo: la richiesta di pull è stata accolta e la correzione è stata integrata rapidamente nel ramo principale della

libreria. In questo modo, il problema di mappatura è stato risolto definitivamente per il progetto e per tutti gli altri utenti della libreria.

Questa esperienza ha chiarito l'importanza delle pratiche open-source, mostrando come la collaborazione e la condivisione diretta delle soluzioni possono velocizzare la risoluzione di problemi complessi e contribuire a migliorare la qualità del software per l'intera comunità.

4.9 Prestazioni e compromessi

4.9.1 Analisi del Ritardo

Durante le prime fasi di sviluppo è stato osservato un ritardo medio di circa cinque secondi tra il completamento di un'attività e l'aggiornamento della lista delle task nel frontend.

- Questo intervallo è legato ai meccanismi interni di sincronizzazione del sistema e al ciclo di polling con Camunda.
- La misurazione è stata ripetuta in più contesti (sviluppo, pre-produzione e produzione), confermando la costanza del ritardo.

4.9.2 Valutazione delle Risorse

Ridurre tale ritardo avrebbe comportato costi significativi in termini di risorse computazionali.

- Un polling più frequente avrebbe incrementato sensibilmente l'uso della CPU lato server, aumentando la pressione sul cluster Kubernetes.
- La memoria necessaria per mantenere connessioni e contesti di esecuzione attivi sarebbe cresciuta proporzionalmente.
- Questo impatto si sarebbe tradotto in una minore efficienza complessiva e in un aumento dei costi infrastrutturali.

4.9.3 Decisione Progettuale

Il team ha scelto consapevolmente di non modificare l'intervallo predefinito di aggiornamento.

- Tale decisione nasce dal bilanciamento tra efficienza delle risorse e percezione dell'utente finale.

- Il ritardo di cinque secondi è stato considerato trascurabile per la maggior parte dei flussi operativi.
- La priorità è stata posta sulla stabilità complessiva del sistema piuttosto che sull'aggiornamento in tempo quasi reale.

4.9.4 Impatto sull'Esperienza Utente

L'analisi dell'esperienza utente ha mostrato che:

- Gli utenti non percepiscono il ritardo come un ostacolo significativo alla produttività.
- Le interazioni con il sistema sono prevedibili e fluide, riducendo la possibilità di confusione o errori dovuti a refresh troppo frequenti.
- Anche su dispositivi con risorse hardware limitate, la scelta progettuale garantisce un'esperienza stabile.

Per concludere, il compromesso trovato tra prestazioni tecniche e usabilità è stato ottimale: un piccolo ritardo, quasi impercettibile dall'utente, consente di ottenere una maggiore efficienza e una gestione dell'infrastruttura più sostenibile.

4.10 Conclusioni

La crescita di *Backend For Frontend* (BFF) è stata lenta ma significativa. Il sistema, inizialmente progettato come una struttura distribuita e modulare, è stato gradualmente modificato per trasformarsi in un servizio distinto, più coerente e pienamente integrato con Camunda 8. Questo cambiamento ha migliorato la sicurezza, l'efficienza e la capacità di governance dei flussi di processo.

I risultati principali possono essere sintetizzati nella seguente sequenza:

- **Da architettura distribuita a servizio unificato:** l'abbandono del modello di microservizi frammentato ha reso la gestione più semplice e la manutenzione più facile, fornendo un unico punto di accesso centralizzato.
- **Integrazione con Camunda 8:** l'implementazione dei client ufficiali (`TaskListClient` e `OperateClient`) ha migliorato l'integrazione con la piattaforma di orchestrazione, rendendo i flussi più sicuri, tipizzati e fluidi.
- **Adozione di strumenti tecnologici avanzati:** la qualità del codice è migliorata, la ridondanza è diminuita e gli sviluppatori sono stati più produttivi grazie all'aggiunta di nuove librerie e pattern architetturali.

- **Sicurezza potenziata:** il BFF garantisce che ogni operazione sia tracciabile e protetta grazie ai controlli avanzati su token JWT, ruoli e compiti assegnati. Ciò riduce il rischio di accesso non autorizzato.
- **Test rigoroso come garanzia di qualità:** il BFF è diventato più affidabile perché fornisce una sorta di documentazione eseguibile del comportamento previsto attraverso test approfonditi basati su mocking e scenari realistici.
- **Scalabilità e resilienza:** il servizio è scalabile e robusto anche quando è centralizzato e può adattarsi a nuove esigenze operative.

Per concludere, la struttura attuale del BFF garantisce la coerenza dei dati, la sicurezza dei flussi e la scalabilità delle operazioni, costituendo un pilastro fondamentale dell'intero ecosistema applicativo. La sua crescita dimostra come un'architettura complessa possa essere trasformata in una soluzione moderna, robusta e pronta ad affrontare le sfide future attraverso scelte progettuali mirate, supportate da strumenti appropriati e da un processo di test organizzato.

Capitolo 5

Conclusioni e Prospettive di Miglioramento

Il presente progetto di tesi è stato sviluppato su commessa di **Wiseside**, per la realizzazione di una web application flessibile e sicura. L'obiettivo principale era quello di fornire uno strumento capace di monitorare in maniera dettagliata il progresso della produzione e della crescita del pollame, offrendo al contempo agli utenti interessati statistiche pertinenti, chiare e facilmente consultabili.

Questa esperienza applicativa ha consentito di affrontare diverse sfide tipiche dello sviluppo software contemporaneo, quali:

- la progettazione di sistemi con un'attenzione particolare agli **aggiornamenti futuri**;
- la **generalizzazione dei componenti** per migliorarne la riusabilità e ridurre le dipendenze;
- l'**integrazione di servizi esterni** in un ecosistema software complesso;
- la gestione di un flusso di lavoro **distribuito tra più persone e sedi geograficamente distanti**.

Il progetto dimostra come un'architettura software ben strutturata possa sostituire efficacemente applicazioni desktop o mobile dedicate e come un approccio modulare, supportato da una documentazione chiara e da test rigorosi, contribuisca a migliorare i processi aziendali e la tracciabilità interna.

5.1 Obiettivi Raggiunti

Per gettare basi solide per una piattaforma completa e scalabile, la tesi ha conseguito i seguenti risultati principali:

- **Analisi approfondita** delle capacità e della struttura delle API offerte da Camunda;
- **Verifica e implementazione dei controlli di sicurezza** necessari a garantire affidabilità e protezione dei dati;
- **Sviluppo di un servizio proxy** flessibile, pensato per essere facilmente scalabile e adattabile a esigenze future;
- **Definizione della modularità** come principio guida nello sviluppo dei diversi servizi costitutivi della piattaforma.

5.2 Lezioni Apprese

La costruzione di un sistema complesso come questo ha fornito insegnamenti preziosi riguardo al delicato equilibrio tra **personalizzazione spinta e manutenibilità**. Infatti, una soluzione completamente su misura, pur rispondendo a esigenze specifiche, non sempre rappresenta la scelta migliore nel lungo periodo. In contesti in cui **scalabilità ed evoluzione** del sistema sono elementi chiave per il successo aziendale, la flessibilità deve essere considerata un requisito imprescindibile. Ciò rende evidente come lo sviluppo di piattaforme a lungo termine debba affrontare la sfida costante di mantenere alti livelli di adattabilità.

5.3 Prospettive di Miglioramento

Sebbene la web application si trovi ancora nelle fasi iniziali del suo ciclo di sviluppo, esistono diversi ambiti nei quali è possibile introdurre miglioramenti significativi:

- **Implementazione di una versione più generica del componente "Complete"**, riducendone la specificità. In questo modo sarà possibile creare nuovi processi su Camunda senza dover intervenire sul *Backend for Frontend* (BFF), aumentando la modularità e riducendo i tempi di configurazione;
- **Introduzione di un sistema di caching più efficiente**, in grado di migliorare le prestazioni complessive rispetto all'approccio attuale e ridurre i tempi di risposta per l'utente finale;
- **Ottimizzazione del numero di chiamate API necessarie** per completare i flussi di lavoro. Una riduzione delle chiamate comporta maggiore velocità di esecuzione e minore carico sui server;
- **Gestione più granulare dei permessi di accesso**, così da garantire che ciascun utente possa accedere soltanto alle API e alle funzionalità coerenti con il proprio ruolo, migliorando sia la sicurezza sia l'esperienza d'uso.

5.4 Considerazioni Finali

In conclusione, questo progetto dimostra come un approccio **metodico e modulare** possa supportare un'azienda nel percorso di trasformazione digitale. Pur avendo raggiunto i principali obiettivi prefissati, la piattaforma mostra ampi margini di crescita e perfezionamento. Un processo di **miglioramento continuo**, alimentato dal feedback degli utenti, rappresenterà un elemento cruciale per una transizione fluida e per il successo a lungo termine. Un ulteriore aspetto di rilievo è la **sostenibilità futura del sistema**, che deve rimanere al centro delle strategie evolutive. Il lavoro svolto fornisce una base solida per lo sviluppo successivo, preparando il terreno per progressi ulteriori e confermando la natura **intrinsecamente evolutiva e trasformativa** del progetto.

Bibliografia

[1] Camunda. *Documentazione di Camunda*. 2025. URL: https://docs.camunda.io/docs/guides/ (cit. a p. xii).

[2] Inc. Pivotal Software. *RabbitMQ Documentation*. 2025. URL: https://www.rabbitmq.com/documentation.html (cit. a p. xii).

[3] Git SCM. *Pro Git Book*. 2025. URL: https://git-scm.com/book/en/v2 (cit. a p. xii).

[4] Spring Framework. *Continuous Integration and Deployment*. 2025. URL: https://spring.io/ (cit. a p. xii).

[5] Oracle. *Java Persistence API Documentation*. 2025. URL: https://docs.oracle.com/javaee/7/tutorial/persistence-intro.htm (cit. a p. xiii).

[6] Douglas Crockford. *The JSON Data Interchange Format*. 2025. URL: https://www.json.org/json-en.html (cit. a p. xiii).

[7] W3C. *Extensible Markup Language (XML) 1.0*. 2025. URL: https://www.w3.org/TR/xml/ (cit. a p. xiii).

[8] OWASP. *Cross-Site Request Forgery (CSRF)*. 2025. URL: https://owasp.org/www-community/attacks/csrf (cit. a p. xiii).

[9] IETF. *JSON Web Token (JWT) RFC 7519*. 2025. URL: https://datatracker.ietf.org/doc/html/rfc7519 (cit. a p. xiii).

[10] GitHub. *GitHub Docs: Issues, Forks, and Pull Requests*. 2025. URL: https://docs.github.com/en/github (cit. a p. xiii).