



UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI MATEMATICA

CORSO DI LAUREA IN  
INFORMATICA

TESI DI LAUREA

## Sperimentazione di Apache Kafka per l'integrazione funzionale di un'applicazione aziendale

Experimenting with Apache Kafka for the Integration of an Enterprise  
Application

*Relatore:*

PROF. TULLIO VARDANEGA

*Laureando:*

ANDREA DORIGO  
1170610

Anno Accademico 2020/2021

# Indice

<b>1</b>	<b>Contesto aziendale</b>	<b>5</b>
1.1	Azienda ospitante . . . . .	5
1.2	Servizi offerti dall'azienda . . . . .	5
1.3	Processi interni e strumenti organizzativi . . . . .	6
1.3.1	Processi interni . . . . .	6
1.3.2	Strumenti organizzativi . . . . .	7
1.4	Dominio tecnologico . . . . .	7
1.4.1	Un dominio flessibile e indipendente . . . . .	7
1.4.2	Sistemi distribuiti . . . . .	8
1.4.3	<i>Service Oriented Architecture<sub>g</sub></i> . . . . .	9
1.4.4	<i>Messaging pattern</i> . . . . .	10
1.4.5	<i>Enterprise Service Bus</i> . . . . .	11
1.4.6	<i>EAI<sub>a</sub></i> e <i>Middleware</i> . . . . .	11
1.4.7	<i>Container</i> e <i>Virtual Machine</i> . . . . .	12
1.4.8	L'introduzione di Apache Kafka . . . . .	15
1.5	L'innovazione all'interno dell'azienda . . . . .	16
<b>2</b>	<b>Apache Kafka nell'Integrazione Aziendale</b>	<b>17</b>
2.1	Obiettivi aziendali . . . . .	17
2.1.1	Migrazione verso un' <i>Event Driven Architecture</i> . . . . .	17
2.1.2	Kafka come Middleware . . . . .	18
2.2	Motivazioni e obiettivi personali . . . . .	19
2.2.1	Scelta del percorso . . . . .	19
2.2.2	Obiettivi personali . . . . .	20
2.3	Il percorso di Stage . . . . .	20
2.3.1	Obiettivi dello <i>stage</i> . . . . .	20
2.3.2	Prodotti attesi . . . . .	20
2.3.3	Contenuti formativi previsti . . . . .	21
2.3.4	Interazione tra studente e referenti aziendali . . . . .	21
2.3.5	<i>Way of working</i> . . . . .	21
2.3.6	Pianificazione del lavoro . . . . .	24
<b>3</b>	<b>Il percorso di stage</b>	<b>27</b>
3.1	Formazione . . . . .	27
3.2	Analisi e modellazione di un caso d'uso . . . . .	28

---

3.3	Progettazione architetturale . . . . .	29
3.3.1	Un <i>middleware</i> basato su un $EDA_a$ . . . . .	29
3.3.2	$UML_a$ <i>sequence diagrams</i> . . . . .	29
3.3.3	$UML_a$ <i>deployment diagrams</i> . . . . .	30
3.3.4	$UML_a$ <i>component diagram</i> . . . . .	31
3.4	Programmazione . . . . .	32
3.4.1	Kafka <i>cluster</i> . . . . .	32
3.4.2	Kafka <i>producer</i> e <i>consumer</i> . . . . .	32
3.4.3	$WS_a$ <i>Client</i> e $WS_a$ <i>Provider</i> . . . . .	33
3.4.4	Protezione dei dati sensibili con Kafka Streams . . . . .	34
3.4.5	Efficienza nello sviluppo . . . . .	34
3.5	Collaudo del prodotto finale . . . . .	35

# Elenco delle tabelle

2.1	Pianificazione settimanale dello <i>stage</i> . . . . .	26
-----	---	----

# Elenco delle figure

1.1	Attuali sedi di Sync Lab . . . . .	6
1.2	Processi interni di cui ho avuto esperienza . . . . .	7
1.3	Illustrazione di un sistema distribuito . . . . .	8
1.4	SOA <sub>a</sub> applicata con un ESB <sub>a</sub> . . . . .	9
1.5	Schema di un <i>messaging design pattern</i> . . . . .	10
1.6	Illustrazione esemplificativa di un ESB <sub>a</sub> . . . . .	11
1.7	Concetti principali del EAI <sub>a</sub> . . . . .	12
1.8	Dalle classiche soluzioni monolitiche ai moderni sistemi a microservizi . . . . .	13
1.9	Differenti implementazioni legate alle VM <sub>a</sub> e <i>container</i> . . . . .	14
1.10	Frammento di codice che espone un esempio di ambiente <i>multi-container</i> con docker-compose . . . . .	14
1.11	Schema di un topic contenente diversi eventi, diviso in partizioni (P), con molteplici <i>producer</i> . . . . .	15
2.1	Illustrazione di un sistema basato sul P2P <sub>a</sub> . . . . .	17
2.2	Illustrazione di un sistema basato sulla EDA <sub>a</sub> . . . . .	18
2.3	Illustrazione di Apache Kafka in un caso d'uso esemplificativo . . . . .	19
2.4	Illustrazione di un sistema a servizi con Kafka . . . . .	20
2.5	<i>Kanban Board</i> del progetto di <i>stage</i> . . . . .	22
2.6	Esempio di un'attività del processo di Formazione . . . . .	23
2.7	Diagramma di Gantt del piano di lavoro . . . . .	26
3.1	Screenshot del corso online <i>acrlongsoa</i> sulla piattaforma Coursera . . . . .	27
3.2	Visione ad alto livello delle differenze tra SOA <sub>a</sub> e microservizi . . . . .	29
3.3	Diagramma di sequenza UML <sub>a</sub> per la re-ingegnerizzazione del flusso asincrono con protezione dei dati sensibili. . . . .	30
3.4	UML <sub>a</sub> <i>deployment diagram</i> per la re-ingegnerizzazione del flusso asincrono (protetto)	31
3.5	UML <sub>a</sub> <i>component diagram</i> per la re-ingegnerizzazione del flusso asincrono (protetto)	31
3.6	JSON <sub>a</sub> inviato al <i>Middleware</i> . . . . .	34
3.7	JSON <sub>a</sub> protetto, ricevuto al termine del <i>callback</i> . . . . .	34
3.8	UML <sub>a</sub> Riassunto dei componenti prodotti nel caso asincrono con <i>callback</i> e della loro comunicazione . . . . .	36

# Capitolo 1

## Contesto aziendale

### 1.1 Azienda ospitante

Sync Lab s.r.l.<sup>1</sup> è un'azienda di produzione software, ICT<sub>a</sub><sup>2</sup> e consulenze informatiche nata nel 2002 a Napoli. L'azienda al suo stato attuale presenta un organico aziendale composto da più di 200 risorse, con un fatturato annuo di 12 milioni, una solida base finanziaria e una diffusione sul territorio a livello nazionale. Sync Lab possiede delle significative fette di mercato riguardanti lo sviluppo di prodotti nel settore mobile, videosorveglianza e sicurezza delle strutture informatiche aziendali.

L'azienda ha acquisito numerose certificazioni ISO LL-C per attestare la qualità dei servizi forniti. La certificazione ISO-9001 attesta la gestione della qualità, ISO-14001 la gestione dell'ambiente, ISO-27001 la sicurezza dei sistemi di gestione dati e ISO-45001 la sicurezza nel luogo di lavoro.

Tra i clienti di Sync Lab vi sono ditte a livello nazionale di grandi dimensioni e ampio organico, come Intesa San Paolo, TIM, Vodafone, Enel e Trenitalia che necessitano prodotti di un'elevata sicurezza e adatti al considerevole flusso di dati aziendale.

Sync Lab ha fornito prodotti e consulenze a più di 150 clienti, distribuiti tra clienti diretti e finali, e attualmente possiede cinque sedi (figura 1.1): Napoli, Roma, Milano, Padova e Verona.

L'azienda è suddivisa in molteplici settori dislocati nelle diverse sedi; l'esperienza personale mi ha portato a conoscere il settore dell'*Enterprise Architecture Integration* e del *Technical Professional Services Padova*.

### 1.2 Servizi offerti dall'azienda

Per comprendere appropriatamente il contesto che ha portato alla nascita del progetto di *stage* è bene conoscere la tipologia di servizi e prodotti che l'azienda offre ai propri clienti. Sync Lab offre per i propri clienti numerosi servizi, tra cui:

- Valutazione e controllo progetti
  - *Planning e project management*; definizione di *Milestone* e *team* di progetto.
  - Valutazione di impatto e *risk analysis*; monitoraggio e *benchmarking*.

---

<sup>1</sup>Sync Lab. URL: <https://www.synclab.it/>.

<sup>2</sup>Information and Communication Technologies



Figura 1.1: Attuali sedi di Sync Lab

Fonte: <https://www.synclab.it/>

- Valutazione e controllo di progetti *software* attraverso l'utilizzo di metriche e modelli economici di stima e previsione.
- Sistemi distribuiti di *Enterprise*
  - Progettazione e realizzazione di sistemi distribuiti *Enterprise* in architettura J2EE<sub>a</sub><sup>3</sup>, EJB<sub>a</sub><sup>4</sup>, COBRA<sub>a</sub><sup>5</sup> e *Web Services*.
  - Progettazione e realizzazione di sistemi basati su MOM<sub>a</sub><sup>6</sup> e JMS<sub>a</sub><sup>7</sup>.
- Tecnologie *Object Oriented*
  - Applicazione delle tecnologie O-O<sub>a</sub> all'analisi e progettazione di *software* applicativo e di sistema e nella definizione di architetture distribuite *enterprise*.
  - Utilizzo di metodologie O-O<sub>a</sub> per progettazione di applicazioni e processi e UML<sub>a</sub>, con supporto di strumenti di *modeling*, applicazione e definizione di *Design Pattern*.

## 1.3 Processi interni e strumenti organizzativi

L'azienda adotta dei processi interni per delineare l'avanzamento di un progetto. Sync Lab utilizza una strategia *Agile* per gestire i progetti in modo da consentire un'evoluzione e adattabilità in base alle richieste del cliente e fornire soluzioni *ad-hoc*.

### 1.3.1 Processi interni

Durante il percorso di *stage* sono stato coinvolto nei processi di Formazione, Progettazione architeturale, Programmazione, Verifica e Collaudo; i processi di Manutenzione ed Evoluzione sono

<sup>3</sup>Java 2 Platform Enterprise Edition

<sup>4</sup>Enterprise Java Bean

<sup>5</sup>Common Object Request Broker Architecture

<sup>6</sup>Message Oriented Middleware

<sup>7</sup>Java Message Service

stati solamente accennati in quanto al di fuori dello scopo del percorso. Questi processi, nella mia esperienza personale, non sono stati delineati rigorosamente: ciò ha lo scopo di garantire libertà e flessibilità allo stagista e di conseguenza all'intero progetto.

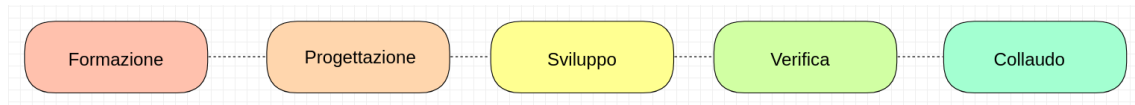


Figura 1.2: Processi interni di cui ho avuto esperienza

Fonte: elaborazione personale

Ogni processo è suddiviso in attività modulari, per rendere l'avanzamento efficace e quantificabile (in figura 1.2 sono illustrati i processi relativi allo *stage* in ordine temporale da sinistra verso destra).

Per il processo di Formazione, Sync Lab fornisce materiale sotto forma di corsi *online* tramite le piattaforme *Coursera*<sup>8</sup> e *Udemy*<sup>9</sup> e diapositive aziendali che illustrano i concetti chiave del settore EAI<sub>a</sub>.

Il processo di Programmazione nella mia esperienza personale è risultato abbastanza libero per quanto riguarda le tecnologie e i *software* utilizzati, purché le scelte fossero adeguatamente motivate e adeguate.

Il processo di Progettazione architetturale è uno dei più complessi, che necessita di una buona dose di esperienza nell'ambito. Per affrontare questo processo, oltre ad approfondire le mie conoscenze riguardo i diversi *design pattern* e *software architecture style*, l'azienda mi ha accompagnato e supportato nella progettazione stessa, con relative motivazioni. Il tutor aziendale Francesco Sanges e il responsabile del settore EAI<sub>a</sub> Salvatore Dore sono stati di fondamentale aiuto in questo processo.

Il processo di Verifica è stato eseguito dal tutor aziendale e dal Responsabile del settore EAI<sub>a</sub> a scadenza settimanale, tramite colloqui *online* o resoconti sulla *online board* di riferimento.

Il processo di Collaudo è avvenuto tramite una presentazione *online* e dimostrazione *live* del prodotto sviluppato all'intera azienda.

### 1.3.2 Strumenti organizzativi

aaaaaaaaa

## 1.4 Dominio tecnologico

### 1.4.1 Un dominio flessibile e indipendente

Il dominio tecnologico aziendale di cui ho avuto esperienza risulta libero e flessibile.

Lo sviluppo del prodotto *software* nell'ambito del EAI<sub>a</sub> è fortemente consigliato essere indipendente dal linguaggio di programmazione, dagli strumenti utilizzati per l'esecuzione e sviluppo, e ove possibile e anche dal Sistema Operativo su cui esso esegue. A tal scopo si utilizzano strumenti quali *Virtual Machine* e *Container*: essi non solo garantiscono l'indipendenza dal Sistema Operativo in uso, ma simulano efficacemente il caso d'uso reale in cui gli eseguibili sono dislocati in più

<sup>8</sup> Coursera | Build Skills with Online Courses. URL: <https://www.coursera.org>.

<sup>9</sup> Online Courses - Learn Anything, On Your Schedule | Udemy. URL: <https://www.udemy.com>.



dispositivi come spesso accade per il cliente finale. Il percorso formativo ha visto l'apprendimento di entrambe le tecnologie tramite l'utilizzo dei *software Virtual Box*<sup>10</sup> e *Docker*<sup>11</sup>, ma infine solo quest'ultima è stata utilizzata durante il progetto.

Nonostante vi sia questa libertà tecnologica riguardo la scelta dei software e linguaggi, è necessario tenere in considerazioni i concetti principali vincolanti del settore, che descrivo in seguito.

### 1.4.2 Sistemi distribuiti

Un concetto fondamentale che caratterizza il dominio tecnologico aziendale, è quello del sistema distribuito, di cui l'azienda ha un forte interesse per soddisfare le necessità di integrazione dei suoi clienti.

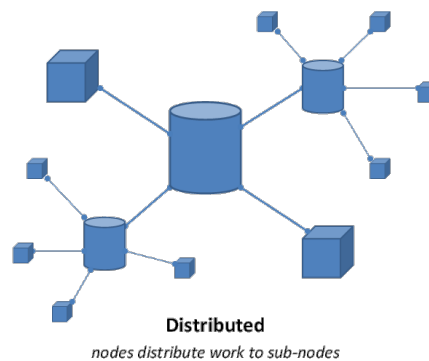


Figura 1.3: Illustrazione di un sistema distribuito

Fonte: <https://www.delphitools.info/DWSH/>

Un sistema distribuito (figura 1.3) è una collezione di componenti indipendenti (spesso collocati in macchine differenti) che condividono dei messaggi fra di loro per raggiungere un obiettivo comune. Un'architettura software basata su di un sistema distribuito necessita di una rete che connette tutti i singoli componenti (macchine, *hardware* o *software*), cosicché sia possibile lo scambio dei messaggi.

Il vantaggio principale di questo tipo di sistemi è la relativamente economica scalabilità orizzontale dei grandi sistemi: per migliorare la *performance* del sistema è sufficiente aggiungere delle nuove macchine, meno costoso che richiedere *hardware* sempre più potente.

Un secondo vantaggio di fondamentale importanza è la tolleranza ai guasti (*fault tolerance*); rispetto ad un sistema centralizzato, ove nel caso di guasto nella macchina centrale si interrompe l'intero sistema, nel caso di un sistema distribuito esso continua a funzionare e fornire il servizio ove vi sia un guasto in un numero (limitato) di macchine.

Un ulteriore punto a favore dei sistemi distribuiti è la bassa latenza: il *service client* si connette al nodo a se più geograficamente vicino, riducendo il tempo di risposta dei sistemi che coprono vasti territori.

Questo tipo di sistemi risulta molto favorevole per i clienti di grande dimensione di Sync Lab, per cui i vantaggi superano lo svantaggio del costo iniziale più alto dato dall'installazione del sistema.

<sup>10</sup> Oracle VM Virtual Box. URL: <https://www.virtualbox.org/>.

<sup>11</sup> Empowering App Development for Developers | Docker. URL: <https://www.docker.com/>.

### 1.4.3 *Service Oriented Architecture<sub>g</sub>*

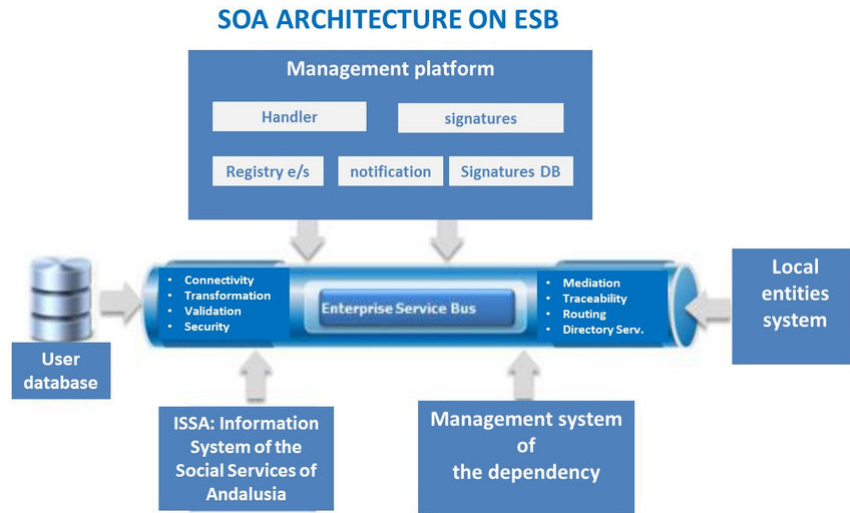


Figura 1.4: SOA<sub>a</sub> applicata con un ESB<sub>a</sub>

Fonte: [https://www.researchgate.net/figure/Fig-9-Service-Oriented-Architecture-SOA-Service-Oriented-Architecture-SOA-Service\\_fig7\\_330599363](https://www.researchgate.net/figure/Fig-9-Service-Oriented-Architecture-SOA-Service-Oriented-Architecture-SOA-Service_fig7_330599363)

La *Service Oriented Architecture<sub>g</sub>* è una tipologia di *software architecture* spesso utilizzata in ambito EAI<sub>a</sub>. Essa definisce un modo per rendere i componenti di un architettura *software* riutilizzabili, tramite una decomposizione di un sistema in parti più piccole che comunicano tramite interfacce di servizio che possono essere classificate come sotto-sistemi. **Ogni servizio in una SOA<sub>a</sub><sup>12</sup> contiene il codice e le integrazioni dei dati necessari per eseguire una funzione aziendale completa e discreta.** Le interfacce di servizio comportano un *loose coupling*, il che significa che possono essere richiamate con poca o nessuna conoscenza della sottostante modalità di implementazione dell'integrazione. I servizi sono esposti utilizzando protocolli di rete standard, come SOAP<sub>a</sub><sup>13</sup>/HTTP<sub>a</sub> o JSON<sub>a</sub><sup>14</sup>/HTTP<sub>a</sub>, per inviare richieste di lettura o modifica dei dati. I servizi sono pubblicati per consentire agli sviluppatori di trovarli rapidamente e riutilizzarli per assemblare nuove applicazioni in modo modulare.

Questo tipo di architettura consente a Sync Lab di creare sistemi basati sui microservizi, derivati dalla SOA<sub>a</sub>, per dare modularità, scalabilità, flessibilità e facilità di manutenzione o aggiornamento ai propri prodotti.

<sup>12</sup> *Service Oriented Architecture<sub>g</sub>*

<sup>13</sup> *Simple Object Access Protocol*

<sup>14</sup> *JavaScript Object Notation*

#### 1.4.4 *Messaging pattern*

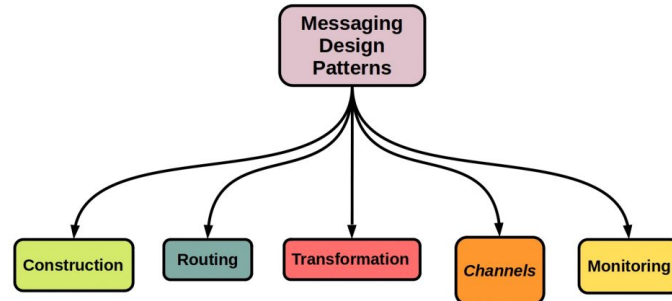


Figura 1.5: Schema di un *messaging design pattern*

Fonte: <https://starship-knowledge.com/messaging-patterns>

I *messaging pattern* sono alla base di ogni sistema di integrazione. Il *design pattern* si occupa dello scambio di messaggi, come si può intuire dal nome. Un messaggio è un pacchetto di dati atomico che può essere trasmesso attraverso un canale in modalità asincrona. Un canale è un condotto virtuale che connette un servizio che invia i dati ad uno che li riceve.

Nella maggior parte dei sistemi di integrazione, il dato potrebbe necessitare diverse elaborazioni e non conoscere direttamente il destinatario del messaggio; è possibile applicare i concetti relativi alla *pipes and filters architecture* inserendo un ricettore comune, un *message router* che si occupa del *routing* di tutti i messaggi e del passaggio di essi attraverso i vari filtri e trasformazioni. Un *message broker* è un modulo *software* spesso posto all'interno di una MOM<sub>a</sub>; si occupa, oltre al *routing*, della validazione, memorizzazione ed invio dei messaggi alle destinazioni appropriate

### 1.4.5 Enterprise Service Bus

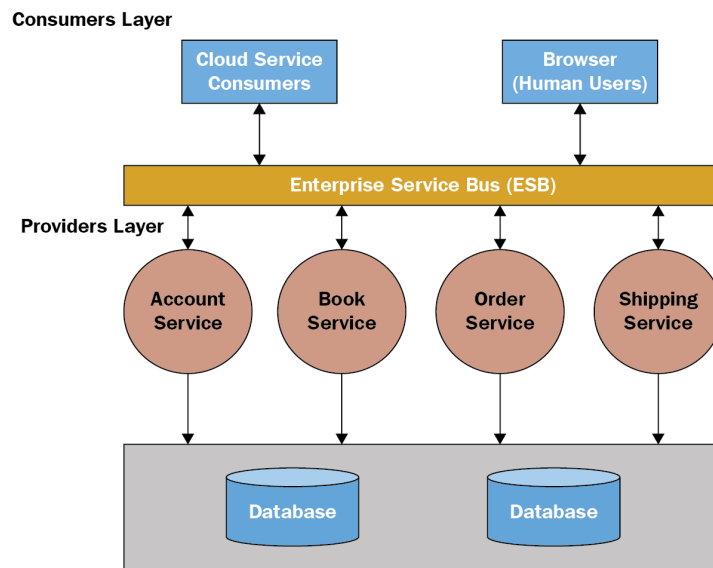


Figura 1.6: Illustrazione esemplificativa di un ESB<sub>a</sub>

Fonte: [https://subscription.packtpub.com/book/application\\_development/9781789133608/1/ch01lv11sec12/service-oriented-architecture-soa](https://subscription.packtpub.com/book/application_development/9781789133608/1/ch01lv11sec12/service-oriented-architecture-soa)

Un ESB<sub>a</sub> è un *pattern* architetturale in cui un *software* centrale consente l'integrazione tra diverse applicazioni (figura 1.6). Esso si occupa della comunicazione, trasformazione e conversione dei dati all'interno di una SOA<sub>a</sub>; è una tipologia più evoluta di *message broker*. Senza strumenti di questo tipo, la *Service Oriented Architecture<sub>g</sub>* porterebbe ad un sistema composto semplicemente da un gruppo di servizi. Ogni servizio dovrebbe occuparsi dello scambio di messaggi con tutti gli altri (P2P<sub>a</sub><sup>15</sup>) creando, nei sistemi più grandi, problemi per quanto riguarda l'estensione e manutenibilità dei servizi.

### 1.4.6 EAI<sub>a</sub> e Middleware

Il percorso di *stage* intrapreso è associato al settore del *Enterprise Architecture Integration*, che si occupa principalmente del EAI<sub>a</sub> (*Enterprise Application Integration<sub>g</sub>*, figura 1.7) ovvero dell'integrazione funzionale di applicazioni aziendali per una clientela di grandi dimensioni (come un'azienda di telecomunicazioni), tramite sistemi di integrazione e *Middleware*.

I *Middleware* e sistemi di integrazione prodotti comprendono l'utilizzo di molteplici linguaggi e tecnologie in continua evoluzione.

Dal sito di Red Hat<sup>16</sup>:

*Il middleware è un software che fornisce alle applicazioni servizi e capacità frequentemente utilizzati, tra cui gestione dei dati e delle API, servizi per le applicazioni, messaggistica e autenticazione.*

*Aiuta gli sviluppatori a creare le applicazioni in modo più efficiente e agisce come un tessuto connettivo tra applicazioni, dati e utenti.*

<sup>15</sup> Point To Point

<sup>16</sup> Cos'è il Middleware? URL: <https://www.redhat.com/it/topics/middleware/what-is-middleware>.

Figura 1.7: Concetti principali del EAI<sub>a</sub>

Fonte: <https://commons.wikimedia.org/wiki/File:KrisangelChap2-EAI.png>

*Può rendere conveniente lo sviluppo, l'esecuzione e la scalabilità di applicazioni alle organizzazioni con ambienti multi cloud e containerizzati.*

I *Middleware* pertanto vedono due importanti utilizzi nel settore EAI<sub>a</sub>:

- **Integrazione su più livelli:** i *Middleware* connettono i principali sistemi aziendali interni ed esterni. Capacità di integrazione quali trasformazione, connettività, componibilità e messaggistica enterprise, abbinate all'autenticazione SSO<sub>a</sub><sup>17</sup>, aiutano gli sviluppatori a estendere tali capacità su diverse applicazioni.
- **Flussi di dati:** le API<sub>a</sub><sup>18</sup> rappresentano una modalità per condividere i dati tra le applicazioni. Un altro approccio è quello del flusso di dati asincrono, che consiste nella replica di un set di dati in un livello intermedio, da cui i dati possono essere condivisi con più applicazioni.
- **Ottimizzazione di applicazioni esistenti:** con l'adozione del *Middleware*, gli sviluppatori possono trasformare le applicazioni monolitiche esistenti in applicazioni *cloud native* o a microservizi, mantenendo i validi strumenti già in uso ma migliorandone prestazioni e portabilità (figura 1.7).

### 1.4.7 Container e Virtual Machine

Come anticipato nella sezione precedente, tra le tecnologie più utilizzate in questo settore aziendale vi sono molte piattaforme che permettono di simulare ambienti distribuiti su più macchine fisiche, ove possibile anche indipendenti dal Sistema Operativo su cui viene eseguito il prodotto software.

La simulazione di questi *distributed environment* avviene grazie a sistemi basati sul concetto di *container* (come Docker e Kubernetes) oppure interi Sistemi Operativi che vengono eseguiti all'interno di una *Virtual Machine*. Il vantaggio principale di queste tecnologie è che rendono

<sup>17</sup> Single Sign On

<sup>18</sup> Application Programming Interface

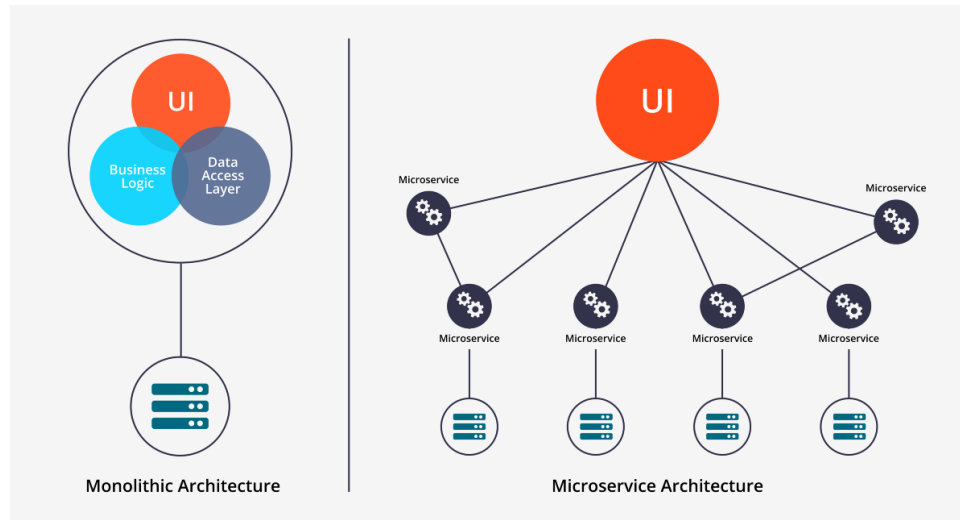


Figura 1.8: Dalle classiche soluzioni monolitiche ai moderni sistemi a microservizi

Fonte: <https://aymax.fr/en/why-a-microservices-architecture/>

l'esecuzione del software al loro interno completamente indipendente dall'ambiente circostante, eliminando problemi di  $OS_a$ <sup>19</sup> differenti tra i componenti del team o tra l'azienda e i clienti o divergenze nelle dipendenze con relative versioni. Un *container* o una *Virtual Machine* contengono tutto il necessario affinché sia possibile eseguire il software al suo interno su diverse macchine fisiche (o anch'esse virtuali).

Queste piattaforme non solo rendono agevole l'esecuzione del software prodotto, ma anche lo sviluppo: la condivisione, *debugging* e manutenzione risultano più agevoli grazie alla condivisione dell'intero *container* o  $VM_a$  con gli altri membri del team.

Inoltre, si adattano particolarmente bene a simulare l'ambiente distribuito, un concetto fondamentale nel settore del *eai*; infatti è sufficiente generare molteplici *container* o  $VM_a$  sulla stessa macchina fisica per simulare un sistema composto da più macchine fisiche distinte, minimizzando l'utilizzo di risorse senza compromettere il risultato del prodotto finale. È così possibile per l'azienda riprodurre un sistema complesso che si avvicina alle risorse ed esigenze effettive del cliente, che usualmente possiede molti computer e server dislocati.

La figura 1.9 rappresenta graficamente le due diverse implementazioni delle due tecnologie. Vi sono dunque delle notevoli differenze, vantaggi e svantaggi tra l'utilizzo dell'una e dell'altra, tra cui:

1. i *container* sono più rapidi delle  $VM_a$  nell'esecuzione;
2. i *container* sono più leggeri delle  $VM_a$  in termini di memoria;
3. i *container* sono più adatti a simulare un'architettura a microservizi, dato la relativa semplicità ed efficienza rispetto ad una  $VM_a$ ;
4. le  $VM_a$  sono considerate tendenzialmente più sicure dei *container*
5. le infrastrutture e strumenti di gestione di grandi quantità di  $VM_a$  sono più consolidate dei corrispettivi strumenti associati ai *container*.

<sup>19</sup> *Operating System*

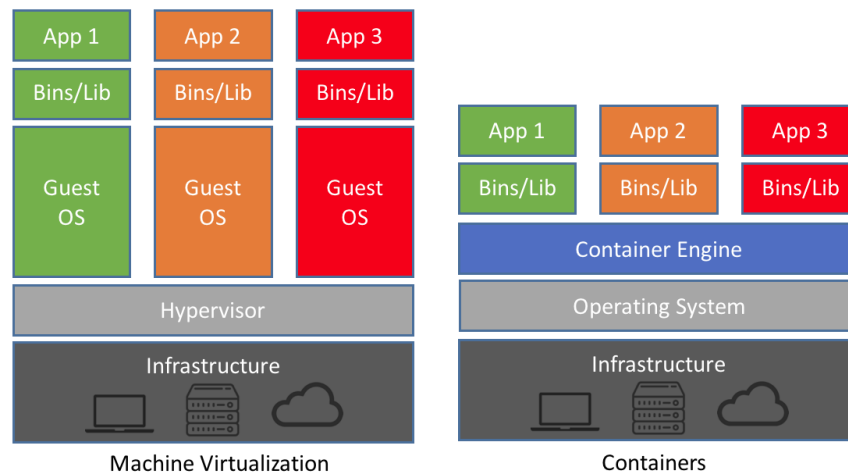


Figura 1.9: Differenti implementazioni legate alle VM<sub>a</sub> e *container*

Fonte: <https://pawsey.sc.github.io/container-workflows/01-docker-intro/index.html>

Durante il percorso di stage ho approfondito le mie conoscenze riguardo entrambe le tecnologie, optando di utilizzare i *container* all'interno del mio progetto di *stage*, poiché più efficiente considerare le risorse a mia disposizione. La scelta della piattaforma di *container* è stata quella di Docker. Più precisamente, ho utilizzato l'estensione *Docker-compose*<sup>20</sup> per gestire in modo elegante la generazione e collaudo di più servizi indipendenti (figura 1.9): non solo questo *software* consente di creare velocemente una rete di *container* comunicanti su di una rete isolata, ma rende anche rapido ed efficiente lo sviluppo grazie alla possibilità di modificare e riavviare rapidamente un singolo servizio.

```

1  version: "3.9"
2  services:
3    web:
4      build: .
5      ports:
6        - "5000:5000"
7      volumes:
8        - ./code
9        - logvolume01:/var/log
10     links:
11       - redis
12     redis:
13       image: redis

```

Figura 1.10: Frammento di codice che espone un esempio di ambiente *multi-container* con docker-compose

Fonte: *elaborazione personale*

Come si può vedere in figura 1.10, è relativamente semplice e veloce creare un ambiente composto da due container (il servizio intitolato **web**, composto a partire dalla cartella corrente e accessibile alla porta 5000, e il servizio **redis**, generato a partire da un'immagine predefinita, entrambi connessi alla stessa rete locale di default di Docker).

<sup>20</sup> Overview of Docker Compose | Docker Documentation. URL: <https://docs.docker.com/compose/>.

Il caso d'uso realizzato nel mio percorso ha simulato grazie ai *container* un sistema a microservizi che simula le risorse di un grande cliente gestore di telecomunicazioni, secondo una visione coerente con il tipo di clientela reale dell'azienda.

#### 1.4.8 L'introduzione di Apache Kafka

In questo periodo Sync Lab sta iniziando dei percorsi per implementare nuove tecnologie nei prodotti *Middleware*. Uno di questi prodotti è Apache Kafka.

Kafka è una piattaforma di *event streaming*, un sistema distribuito e moderno basato sugli eventi anziché su di una soluzione più classica come può essere quella del *request/response* e P2P<sub>a</sub>.

L'*event streaming* è una pratica focalizzata sul catturare dati in *real-time* da diverse fonti come *database*, sensori e dispositivi mobili sotto la forma di un flusso continuo di eventi; garantisce uno scambio continuo di informazioni e la loro interpretazione.

Apache Kafka è un *software* fondato sul *design pattern* del *publish/subscribe*. Contiene infatti delle API<sub>a</sub> intitolate *Producer* e *Consumer*, che un utente della piattaforma può utilizzare rispettivamente per pubblicare degli eventi o riceverli istantaneamente. Permette inoltre di memorizzare questo flusso di dati in modo affidabile, *fault tolerant*, e duraturo, o di processare il flusso per processarlo e trasformarlo.

Un evento è definito come un'occorrenza significativa o un cambiamento di stato del sistema, e nell'ottica del *messaging design pattern* occuperebbe il ruolo di un messaggio atomico.

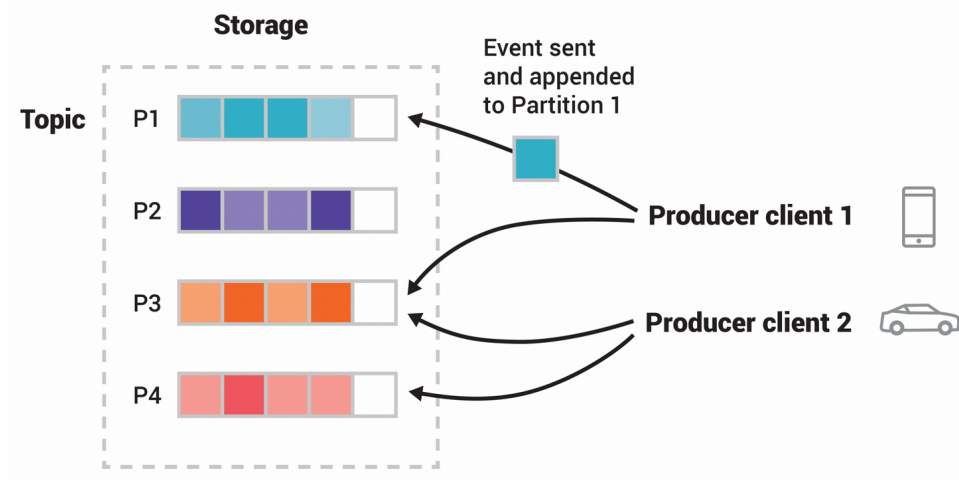


Figura 1.11: Schema di un topic contenente diversi eventi, diviso in partizioni (P), con molteplici *producer*

Fonte: <https://kafka.apache.org/intro>

Essi sono organizzati all'interno di *topics*, delle liste ordinate di eventi in cui molteplici applicazioni possono scrivere o leggere tali eventi simultaneamente. In figura si può vedere un esempio di *topic*, diviso in diverse partizioni, in cui due *producer* inseriscono dati simultaneamente.

Kafka viene eseguito come un gruppo (*cluster*) di *server* distribuiti. Alcuni di questi *server* sono chiamati *broker*, e insieme compongono il livello di memorizzazione (*storage layer*). Altri *server* sono dedicati all'esecuzione di Kafka Connect, un componente essenziale per l'importazione dei dati da altre fonti; esso è molto rilevante nel campo dei *Middleware*, poiché permette di integrare Kafka all'interno di sistemi pre-esistenti in modo graduale, con un investimento iniziale parziale.



I *client* invece permettono di sviluppare applicazioni distribuite a microservizi che leggono, scrivono e processano il flusso di dati; questi *client* mettono a disposizione delle interfacce per molti linguaggi e *software* differenti, tra cui Java, Scala, Python, C++/C oltre a delle REST<sub>a</sub> API<sub>a</sub>.

Il sistema è molto adattabile, dato che può essere installato anche su *acrlongvm*, *container*, ambienti *cloud* e addirittura *bare-metal hardware*. La piattaforma è costituita da un sistema distribuito di *server* e *client* che comunicano attraverso un protocollo *network* TCP<sub>a</sub><sup>21</sup> ad alta *performance*.

Il *software* ha dimostrato negli anni recenti un notevole successo in diversi campi<sup>22</sup>, come quello del flusso di *Big Data*, del monitoraggio e dell'elaborazione dati in tempo reale. L'adozione del *software* nell'ambito del EAI<sub>a</sub> è in crescita dato le dimostrate qualità nel gestire grandi moli di dati: la sua performance, sicurezza e scalabilità sono i punti che hanno portato il software al suo attuale successo.

## 1.5 L'innovazione all'interno dell'azienda

Questo contesto dell'integrazione aziendale porta dunque l'impresa ad avere un'importante propensione all'innovazione, talvolta esplicitamente richiesta dai clienti.

Una direzione dell'evoluzione attuale nel settore EAI<sub>a</sub> riguarda la migrazione verso sistemi sempre più distribuiti, in accordo con l'ambiente di lavoro descritto nelle sezioni precedenti, in grado di gestire efficacemente ed in tempo reale flussi di dati in continua crescita e appartenenti al mondo del *Big Data*.

L'avanguardia tecnologica è pertanto uno dei principali temi dell'azienda, che garantisce che essa rimanga sempre competitiva sul mercato dei sistemi di integrazione e nel settore dell'EAI<sub>a</sub>. L'implementazione di nuove tecnologie, come può essere una *Event Driven Architecture* basata su Kafka, permetterebbe a Sync Lab di offrire soluzioni sempre più moderne, scalabili e affidabili ai propri clienti.

---

<sup>21</sup> *Transmission Control Protocol*

<sup>22</sup>Fonte: <https://kafka.apache.org/powered-by>

## Capitolo 2

# Apache Kafka nell'Integrazione Aziendale

### 2.1 Obiettivi aziendali

#### 2.1.1 Migrazione verso un'Event Driven Architecture

Per soddisfare le richieste dei clienti ed essere sempre competitiva e all'avanguardia, una priorità di Sync Lab sono le esplorazioni tecnologiche e di prodotto anche tramite l'utilizzo di percorsi di *stage* insieme ai laureandi, come quanto accaduto nella mia esperienza. Questi percorsi consentono all'azienda non solo di testare l'utilizzo di nuovi *software* ma anche di conoscere e mettere alla prova le capacità del laureando in vista di una potenziale assunzione al termine dello *stage*.

Nel settore dell'*Enterprise Application Integration<sub>g</sub>*, l'evoluzione tecnologica è diretta verso soluzioni sempre più distribuite e con un flusso di dati in continuo aumento. Uno degli obiettivi specifici nell'area *EAI<sub>a</sub>* di Sync Lab è pertanto quello di trovare un *software* o tecnologia in grado di soddisfare i bisogni dei clienti di gestire un flusso di dati di dimensioni molto maggiori a quelle attuali, tramite architetture a messaggio che utilizzano servizi distribuiti.

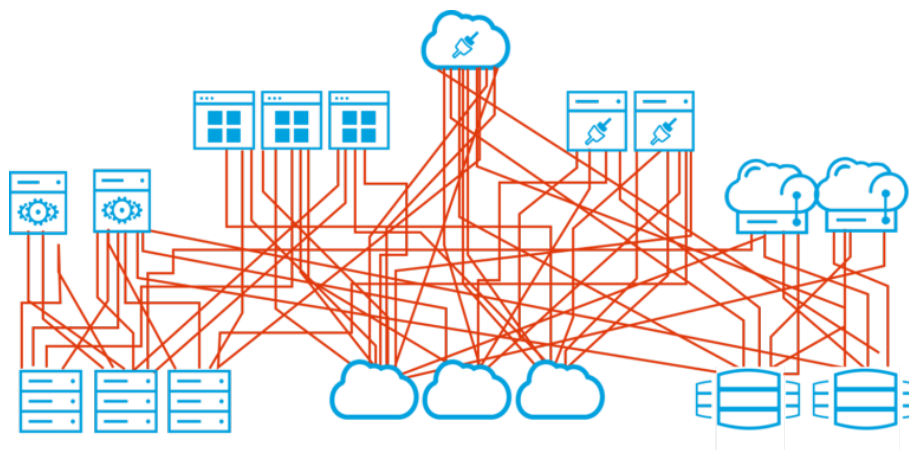


Figura 2.1: Illustrazione di un sistema basato sul P2P<sub>a</sub>

Fonte: <https://news.pwc.be/messaging-architecture-with-salesforce/>

Nei *Middleware* per i sistemi di integrazione, l'aumento del flusso di dati e lo spostamento verso strutture distribuite provoca una difficoltà nella trasmissione dei dati tra i diversi servizi. Nell'ambito dell'integrazione per un cliente di piccole dimensioni, in cui i dati circolano tra un numero di componenti limitato, può essere sufficiente un'architettura di tipo P2P<sup>1</sup>.

Nel caso di un cliente di maggiori dimensioni tuttavia questo approccio rende la manutenzione e gestione del flusso di dati molto difficoltoso e costoso in termini di risorse (figura 2.1), dato che il grande numero di collegamenti tra i vari punti.

Una delle soluzioni che viene maggiormente implementata per risolvere questo problema è la migrazione verso una EDA<sub>a</sub> (*Event Driven Architecture*), un'architettura basata sugli eventi in grado di scambiare dati tra punti multipli.

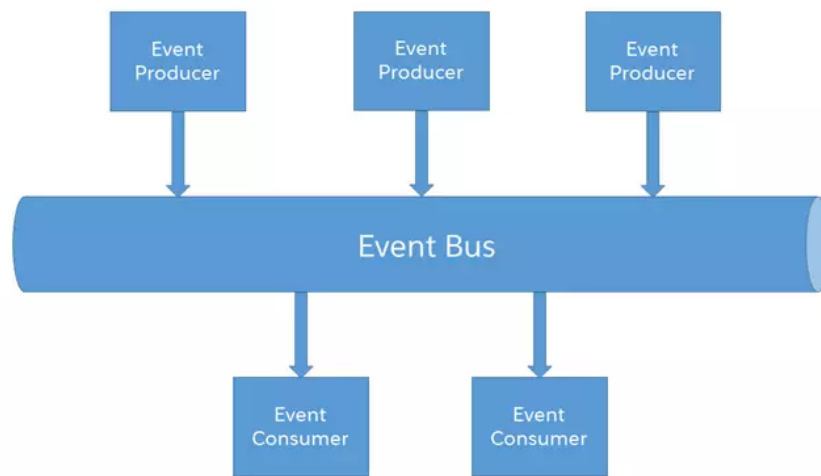


Figura 2.2: Illustrazione di un sistema basato sulla EDA<sub>a</sub>

Fonte: <https://news.pwc.be/messaging-architecture-with-salesforce/>

Questo tipo di architettura è pertanto definita per gestire la produzione, il rilevamento e la reazione agli eventi (figura 2.2) grazie ad un *Design Pattern* di tipo *Publish/Subscribe*, eliminando i problemi visti precedentemente causati dal sistema P2P<sub>a</sub>. Questa architettura prevede l'utilizzo di servizi chiamati *Producer*, il cui scopo è fornire dati al *event bus* centrale. Una volta che i dati sono inseriti all'interno del *bus* centrale, ogni servizio in ascolto (*Subscriber*) li riceverà idealmente in tempo reale.

### 2.1.2 Kafka come Middleware

Per soddisfare le esigenze di innovazione l'azienda ha avviato un percorso per indagare le capacità del software Apache Kafka nell'ambito dell'integrazione aziendale.

Apache Kafka si integra ottimamente in molti sistemi basati sul *messaging pattern* e una EDA<sub>a</sub>, in cui lo scambio affidabile di dati tra numerosi servizi in tempo reale è essenziale (in figura 2.3 è illustrato un caso d'uso esemplificativo di un sistema distribuito basato su Kafka).

L'interesse di Sync Lab nel *software* risiede dunque nell'utilizzo di Kafka come un *Middleware* per soddisfare i problemi di integrazione aziendale e re-ingegnerizzare i flussi di dati preesistenti,

<sup>1</sup> *Point To Point*

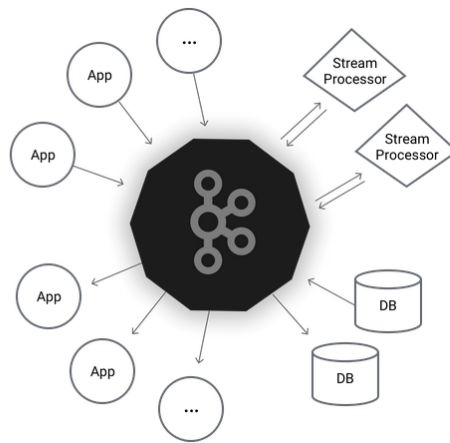


Figura 2.3: Illustrazione di Apache Kafka in un caso d'uso esemplificativo

Fonte: <https://iotbyhvm.ooo/apache-kafka-a-distributed-streaming-platform/>

ovvero sviluppare un nuovo sistema che consenta la comunicazione tra differenti servizi con un rapido flusso di dati fra di essi.

L'azienda ha avviato un percorso per testare le capacità di Kafka rispetto agli attuali strumenti utilizzati nel settore, per valutare i vantaggi e svantaggi che l'adozione di tale *software* può fornire al cliente.

Sono numerosi i vantaggi che Kafka può portare nel settore, fra cui:

- gestione rapida e performante di un enorme flusso di dati;
- scalabilità;
- sicurezza riguardo la persistenza dei dati;
- semplice integrazione e affiancamento a sistemi già esistenti;
- l'essere una piattaforma *open source*;
- processazione dei dati in tempo reale integrata.

## 2.2 Motivazioni e obiettivi personali

### 2.2.1 Scelta del percorso

Una delle ragioni che mi ha portato a scegliere questo percorso di *stage* è l'interesse verso Apache Kafka. L'utilizzo della piattaforma di *event streaming* è sempre più in crescita, come l'evoluzione verso sistemi sempre più distribuiti e a microservizi.

Un altro fattore fondamentale alla scelta del percorso sono stati la familiarità con l'azienda, il personale giudizio positivo che ho avuto riguardo il loro metodo di lavoro e la libertà di sviluppo concessa: ho ritenuto importante la possibilità di elaborare personalmente un'architettura del caso d'uso con una visione ad alto livello, anziché il semplice sviluppo di un software predeterminato e dal percorso strettamente imposto.

### 2.2.2 Obiettivi personali

L'obiettivo fondamentale dello *stage* è colmare il divario tra il mondo accademico e quello lavorativo. Grazie al percorso di *stage* in una ditta esterna ho avuto l'opportunità di conoscere l'ambiente di lavoro di un'azienda nel campo ICT<sub>a</sub>, facilitandomi l'inserimento nel mondo del lavoro.

Un altro obiettivo è ottenere una formazione riguardo la tecnologia di Kafka, che ritengo possa arricchire fortemente le mie capacità e *skill* professionali. Sono pertanto interessato a sviluppare la mia formazione riguardo l'utilizzo e le implicazioni di questa tecnologia in rapida espansione, la cui formazione potrà essermi utile in molti campi anche al di fuori degli obiettivi dell'azienda ospitante lo *stage*.

## 2.3 Il percorso di Stage

### 2.3.1 Obiettivi dello *stage*

Il percorso di *stage* offerto dall'azienda si inserisce all'interno della strategia aziendale più ampia descritta sopra. Al fine di esplorare la tecnologia di Apache Kafka nell'ambito di un *Middleware* per l'integrazione aziendale, l'azienda ha proposto un percorso di ***stage* il cui obiettivo è la re-ingegnerizzazione di un flusso di dati asincrono, utilizzando un'architettura basata su Kafka all'interno di un caso d'uso simulato tramite servizi indipendenti.**

Lo stagista ha il compito di osservare, testare e verificare che il *software* possa svolgere alcuni compiti inerenti all'area del EAI<sub>a</sub>, analizzando alcuni casi d'uso presenti in un *Middleware* aziendale in ambito *telco*. Il percorso di prevede una durata di 300 ore lavorative.

### 2.3.2 Prodotti attesi

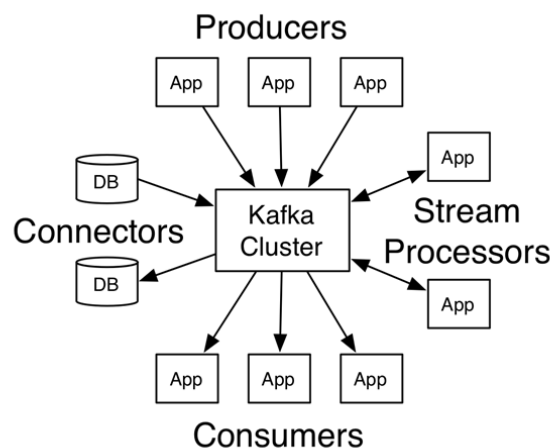


Figura 2.4: Illustrazione di un sistema a servizi con Kafka

Fonte: <https://kafka.apache.org/20/documentation.html>

I prodotti attesi al termine dello *stage* sono dunque associati alla realizzazione di tre flussi di integrazione, basati su dei casi d'uso reali, per la gestione dei paradigmi di integrazione asincrono e asincrono con *callback* (due requisiti obbligatori), e sincrono ove fosse disponibile del tempo

aggiuntivo e se ritenuto opportuno; durante il percorso considerato il contesto e le opportunità offerte dal *software* questo obiettivo verrà sostituito per testare delle funzionalità aggiuntive di Kafka.

Il prodotto *software* finale sarà un sistema basato su servizi indipendenti costruito con un'architettura di tipo EDA<sub>a</sub> tramite l'utilizzo di Kafka (figura 2.4).

### 2.3.3 Contenuti formativi previsti

La realizzazione di questi prodotti necessita una sostanziale formazione dello stagista riguardo i principali concetti del settore del *Enterprise Application Integration* e l'utilizzo della piattaforma di *event streaming* Kafka. Più precisamente, i contenuti formativi previsti durante questo percorso di *stage* sono i seguenti:

- Concetti chiave del *Enterprise Application Integration*<sub>g</sub>;
- *Design architetturali*;
- Cenni di *Networking* applicato alle architetture distribuite;
- Architetture di Integrazione e *Middleware*;
- Apache Kafka.

### 2.3.4 Interazione tra studente e referenti aziendali

Regolarmente, (almeno una volta la settimana) ci saranno incontri online (tramite la piattaforma Google Meet) con il tutor aziendale Francesco Giovanni Sanges, il responsabile dell'area EAI<sub>a</sub> Salvatore Dore e gli esperti delle tecnologie affrontate. I meeting saranno necessariamente online, dato il dislocamento dei vari membri in diverse città.

Lo scopo di questi incontri sarà quello di verificare lo stato di avanzamento, chiarire gli obiettivi ove necessario, affinare la ricerca e aggiornare la pianificazione iniziale.

### 2.3.5 *Way of working*

All'inizio del percorso ho delineato un *way of working*, ovvero un metodo di lavoro da mantenere per tutta la durata dello *stage*, insieme al tutor aziendale, il responsabile dell'area EAI<sub>a</sub> e gli esperti del settore.

Per garantire un buon livello organizzativo, quantificare l'avanzamento e rendere agevole la verifica e il supporto tecnico l'azienda ha proposto l'utilizzo di una *board* di progetto.

L'organizzazione efficiente del progetto è dunque garantita dall'utilizzo dei vari strumenti a supporto, quali *Kanban Board* (come *Click Up*<sup>2</sup> per la gestione di progetto e *Notion*<sup>3</sup> per le prenotazioni della postazione di lavoro in sede), *chat* (come *Google Chat*<sup>4</sup>) per i confronti rapidi con gli altri membri interni al progetto ed e-mail per le comunicazioni con componenti esterni al progetto.

Lo strumento più utilizzato in ambito organizzativo durante il mio percorso è la *Kanban Board* di *Click Up*, che ha permesso la gestione, il confronto, la quantificazione e la verifica del progresso.

<sup>2</sup> *ClickUp*<sup>TM</sup> | One app to replace them all. URL: <https://clickup.com>.

<sup>3</sup> *Notion* - The all-in-one workspace for your notes, tasks, wikis, and databases. URL: <https://www.notion.os>.

<sup>4</sup> *Google Chat*. URL: <https://chat.google.com>.

Tra le tante opzioni disponibili *ClickUp* possiede numerosi vantaggi rispetto la concorrenza: la piattaforma è ricca di funzionalità, pulita nell'esposizione dello stato del progetto, e la maggior parte delle sue funzioni sono gratuite. La figura seguente illustra, a titolo esemplificativo, uno *screenshot* che raffigura lo stato dell'avanzamento.

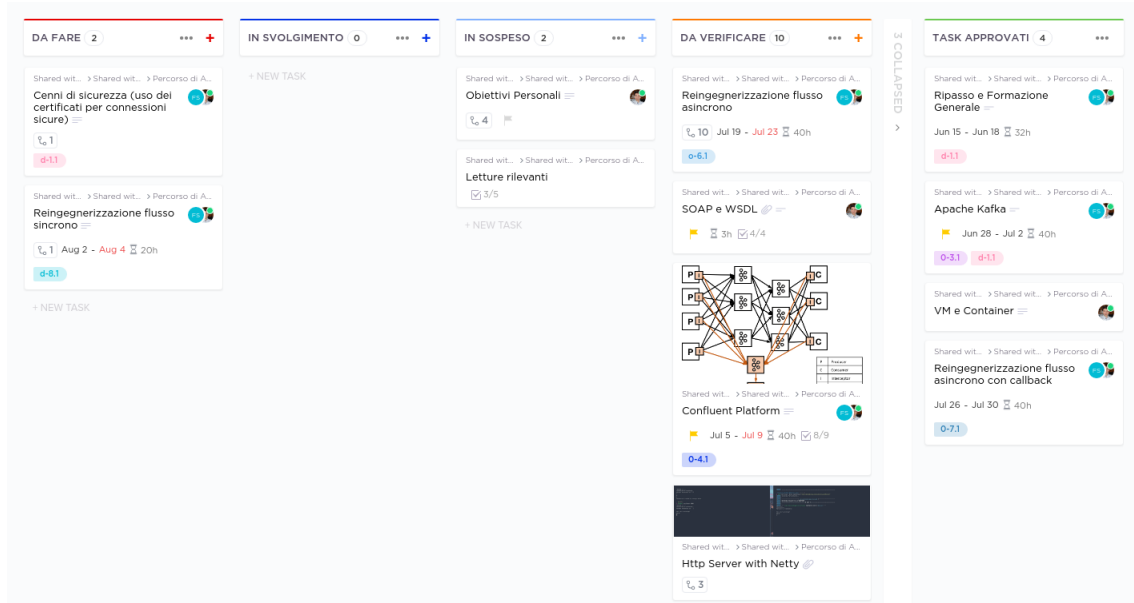


Figura 2.5: *Kanban Board* del progetto di *stage*

Fonte: elaborazione personale

Le attività (*task*) vengono inizialmente create nella colonna "DA FARE" dal tutor aziendale o dal sottoscritto, ove ritenuto opportuno. Per dimostrare l'avanzamento il *task* si sposta verso destra a seconda dello stato raggiunto; lo stagista ha la responsabilità del cambiamento di stato fino alla colonna "DA VERIFICARE", dopodiché è compito del tutor aziendale la verifica e lo spostamento del *task* in "TASK APPROVATI", che comporta l'approvazione finale e conclusione dell'attività.

Per tenere traccia del lavoro svolto riguardante una specifica attività ho utilizzato le *card* messe a disposizione dalla piattaforma, che mi hanno consentito di delineare precisamente la pianificazione e descrizione dell'avanzamento in dettaglio del singolo *task*.

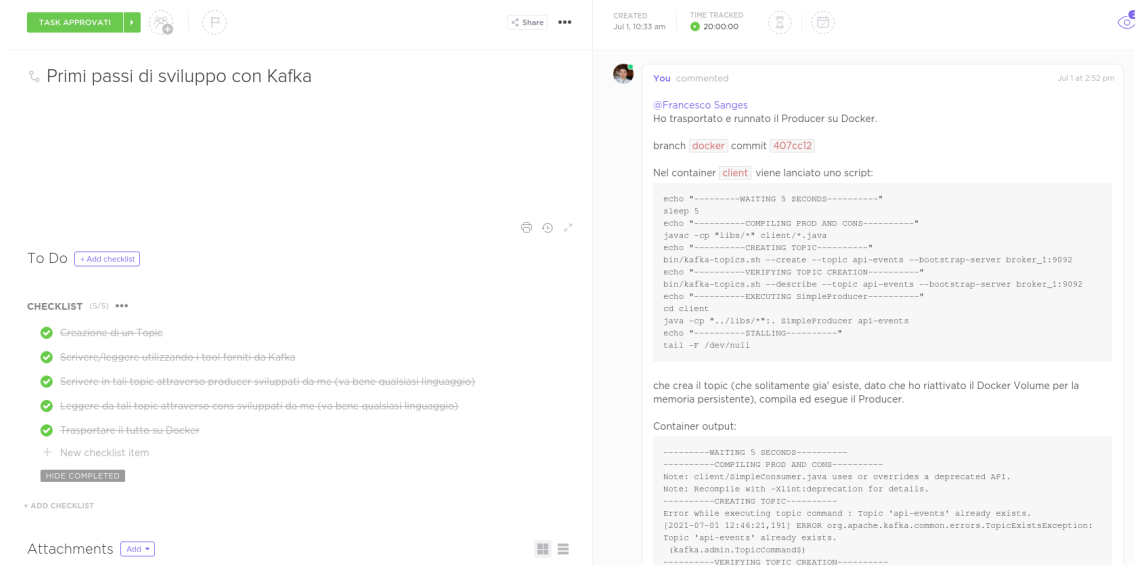


Figura 2.6: Esempio di un'attività del processo di Formazione

Fonte: elaborazione personale

Questa *card* contiene una casella di testo per inserire una descrizione e appunti utili ove sia richiesto, una *checklist* approfondita, e una colonna che mantiene uno storico dei commenti; quest'ultima colonna non solo permette a me di mantenere un'importante resoconto sul lavoro svolto, ma consente anche al tutor aziendale e esperti del settore di quantificare il progresso e di fornire un aiuto rapido e contestuale.

Per la condivisione di codice è stato possibile utilizzare uno strumento a mia discrezione, e pertanto ho utilizzato Git per creare una *repository* e successivamente ne ho fatto l'*upload* in modalità privata (secondo indicazioni aziendali) su Github. L'inserimento del codice identificativo del *commit* all'interno di un commento allo scopo di *log* ha favorito ulteriormente il tracciamento del progresso e reso agevole l'eventuale supporto da parte degli esperti aziendali.

All'inizio del percorso il tutor aziendale e il responsabile del EAI<sub>a</sub> hanno creato delle *card* contenenti le attività previste per ogni settimana (*task*) al fine di fornire una struttura generale del progetto. All'interno di questi *task* vi sono i concetti chiave, attività previste e obiettivi settimanali che lo stagista è tenuto a seguire per garantire l'efficacia del prodotto finale. Oltre a questi *task* principali, ho potuto creare di *task* ausiliari e dei *sub-task* per descrivere più adeguatamente l'attività in corso.

Ciascun *task* contiene una colonna laterale dove ho mantenuto un *log* di tutto ciò che è stato eseguito relativo al *task* in questione, allo scopo di esplicitarne il progresso e rendere agevole un eventuale supporto dal tutor o l'evoluzione futura.

Ogni settimana è previsto un *online meeting* per la verifica del progresso ove necessario, la risposta ad eventuali questioni sollevate, e spiegazioni riguardo lo sviluppo della settimana successiva. Alcune di queste videoconferenze ha visto la partecipazione di altri esperti che mi hanno aiutato a comprendere meglio il caso d'uso da re-ingegnerizzare, riassumendo lo stato attuale del sistema d'integrazione per uno dei clienti con relativi *file* utilizzati.

Per mantenere alto il livello di organizzazione, efficienza ed efficacia, all'inizio di ogni giornata lavorativa ho creato un breve piano giornaliero con successivo consuntivo a fine giornata. Questo



ha permesso al tutor di verificare rapidamente il corretto avanzamento del processo in corso e a me di mantenere il *focus* su di esso.

### 2.3.6 Pianificazione del lavoro

Ad ogni incremento è associato un requisito obbligatorio, desiderabile o facoltativo. A questi requisiti vi è associato un codice identificativo per favorirne il tracciamento futuro, in che precede la voce descrittiva dell'incremento. Ogni codice è composto da una lettera seguita da dei numeri interi, secondo il seguente modello:

**A-X.Y.Z**

ove, da sinistra verso destra:

- **A** rappresenta la lettera che qualifica il requisito come obbligatorio, desiderabile o facoltativo, secondo la seguente notazione:
  - *O* per i requisiti obbligatori, vincolanti in quanto obiettivo primario richiesto dal committente;
  - *D* per i requisiti desiderabili, non vincolanti o strettamente necessari, ma dal riconoscibile valore aggiunto;
  - *F* per i requisiti facoltativi, rappresentanti valore aggiunto non strettamente competitivo.
- **X** rappresenta la settimana in cui viene inizialmente pianificato l'incremento (identificata da un numero incrementale e intero, partendo da 1). Questo consente allo studente, al tutor interno e al tutor esterno una rapida quantificazione dell'avanzamento corrente dello stage rispetto a quanto inizialmente pianificato.
- **Y** rappresenta la posizione sequenziale prevista dell'incremento all'interno della settimana (incrementale e intero, partendo da 1). Esso è strettamente associato alla lettera.

Di seguito viene presentata la pianificazione settimanale delle ore lavorative previste. Ad ogni settimana sono assegnate le voci contenenti gli incrementi previsti in essa, ove i codici utilizzano la notazione descritta precedentemente.

Tutte le settimane prevedono 40 ore lavorative, fatta eccezione per l'ultima che ne prevede 20.

SETTIMANA	CODICE	TASK ASSOCIATI
	O-1.1	Incontro con le persone coinvolte nel progetto per discutere i requisiti e le richieste relative al sistema da sviluppare
	O-1.2	Verifica credenziali e strumenti di lavoro assegnati
	O-1.3	Presenza visione dell'infrastruttura esistente

	D-1.1	<p>Ripasso approfondito riguardo i seguenti argomenti:</p> <ul style="list-style-type: none"> <li>• Ingegneria del <i>software</i>;</li> <li>• Sistemi di versionamento;</li> <li>• Architetture <i>software</i>;</li> <li>• Cenni di <i>Networking</i>.</li> </ul>
2	O-2.1	Nozioni fondamentali riguardo EAI <sub>a</sub> e SOA <sub>a</sub> <sup>5</sup>
	O-2.2	<p>Approfondimenti riguardo le Architetture a Messaggio, in particolare:</p> <ul style="list-style-type: none"> <li>• <i>Integration Styles</i>;</li> <li>• <i>Channel Patterns</i>;</li> <li>• <i>Message Construction Patterns</i>;</li> <li>• <i>Routing Patterns</i>;</li> <li>• <i>Transformation Patterns</i>;</li> <li>• <i>System Management Patterns</i>.</li> </ul>
3	O-3.1	<p>Apache Kafka:</p> <ul style="list-style-type: none"> <li>• Introduzione a Kafka;</li> <li>• Concetti fondamentali di Kafka;</li> <li>• Avvio e CLI<sub>a</sub><sup>6</sup>;</li> <li>• Programmazione in Kafka con Java.</li> </ul>
	D-3.1	Esempi e applicazioni di Apache Kafka
4	O-4.1	<p>Confluent Platform:</p> <ul style="list-style-type: none"> <li>• <i>Service registry</i>;</li> <li>• REST<sub>a</sub> <i>proxy</i>;</li> <li>• kSQL;</li> <li>• Confluent <i>connectors</i>;</li> <li>• <i>Control center</i>.</li> </ul>
5	O-5.1	Analisi dei casi d'uso reali
	O-5.2	Realizzazione dei componenti per l'esecuzione dei casi di test
6	O-6.1	Analisi re-ingegnerizzazione e collaudo del flusso di integrazione asincrono
7	O-7.1	Analisi e re-ingegnerizzazione e collaudo del flusso di integrazione asincrono con callback

<sup>5</sup> *Service Oriented Architecture*<sup>6</sup> *Command Line Interface*

8	O-8.1	Analisi e re-ingegnerizzazione e collaudo del flusso di integrazione sincrono
---	-------	---

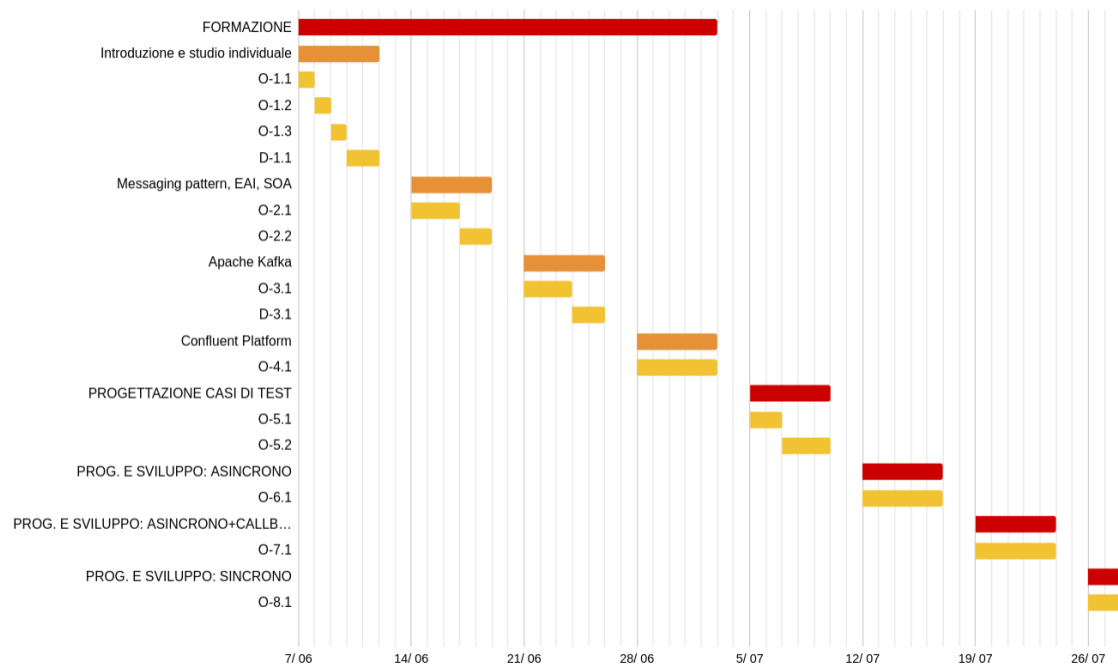
Tabella 2.1: Pianificazione settimanale dello *stage*

Figura 2.7: Diagramma di Gantt del piano di lavoro

Fonte: elaborazione personale

Secondo questa pianificazione, (di cui la figura 2.7 rappresenta il diagramma di Gantt) le 300 ore di *stage* previste sono approssimativamente divise in:

- 160 ore di Formazione sulle tecnologie;
- 60 ore di Progettazione dei componenti e dei test;
- 60 ore di Sviluppo dei componenti e dei test;
- 20 ore di Valutazioni finali, Collaudo e Presentazione della Demo.

## Capitolo 3

# Il percorso di stage

### 3.1 Formazione

Il processo di Formazione ha avuto un importante ruolo all'interno dello *stage*, con una durata complessiva di circa quattro settimane. La causa di questo lungo periodo di formazione è data dallo studio di diversi ambiti e concetti per me nuovi, in particolare il settore del *Enterprise Application Integration<sub>g</sub>* e la tecnologia di Kafka.

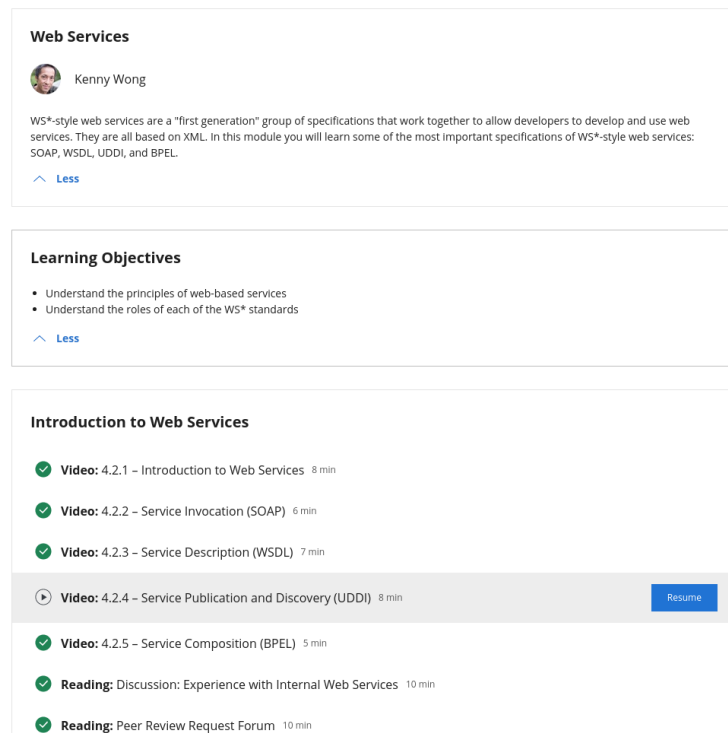


Figura 3.1: Screenshot del corso online *acrlongsoa* sulla piattaforma Coursera

Fonte: elaborazione personale

Durante questo processo Sync Lab mi ha fornito del materiale didattico per l'apprendimento, quali diapositive e appunti di origine aziendale e l'accesso a dei corsi riguardo *Software architecture*,

*Service Oriented Architecture<sub>g</sub>* (figura 3.1) (tramite i corsi online su *Coursera*) e Apache Kafka (tramite i corsi online su *Udemy*).

Il tutor aziendale e responsabile EAI<sub>a</sub> hanno fornito durante i *meeting* settimanali ulteriori chiarimenti e approfondimenti sul come Sync Lab applica questi concetti nello sviluppo di architetture *software*.

Per i corsi online a maggiore contenuto nozionistico ho redatto degli appunti riassuntivi, con lo scopo di consolidare l'apprendimento e velocizzare la verifica del tutor aziendale.

## 3.2 Analisi e modellazione di un caso d'uso

Ad alcune videoconferenze ha partecipato anche un esperto *senior* aziendale, esterno al progetto di *stage* in questione, per illustrarmi un caso d'uso in cui l'azienda ha esposto un prototipo di sistema di integrazione utilizzando i concetti di *Web Service*, SOAP<sub>a</sub><sup>1</sup> e *request/response*, permettendomi la visualizzazione dei file WSDL<sub>a</sub><sup>2</sup>, XML<sub>a</sub><sup>3</sup> e XSD<sub>a</sub><sup>4</sup> associati. Ho pertanto generato un caso d'uso adatto agli scopi dello *stage* ispirandomi al caso d'uso reale illustrato.

Il caso d'uso modellato tratta una richiesta di credito telefonico da parte di un cliente ad un'azienda di telecomunicazioni tramite *Web Service*, per soddisfare il requisito dello sviluppo della re-ingegnerizzazione del flusso di dati asincrono. La *request* avviene tramite flusso di un file JSON<sub>a</sub><sup>5</sup> che viene trasmesso attraverso i vari servizi che compongono il sistema di integrazione, basato sul *Design Pattern* di tipo *publish/subscribe*.

Va precisato che il contenuto di tale JSON<sub>a</sub> non è strettamente rilevante allo sviluppo e funzionamento del *Middleware*, ma aiuta a stabilire il contesto di utilizzo.

Il caso d'uso è composto dai seguenti passaggi:

1. il cliente (*WS<sub>a</sub><sup>6</sup> Client*) effettua una richiesta di credito tramite invio di un file JSON<sub>a</sub> al successivo Servizio Web in ascolto.
2. il servizio composto da REST<sub>a</sub><sup>7</sup> WS<sub>a</sub> e *Request Producer* riceve il JSON<sub>a</sub> e lo inserisce in Kafka tramite l'apposito *Kafka Producer*, assumendo la funzione di *publisher*.
3. il servizio di *Request Consumer*, sottoscritto al *topic* in questione riceve il JSON<sub>a</sub> e lo invia al WS<sub>a</sub> finale tramite una REST<sub>a</sub> request.
4. il servizio in coda chiamato WS<sub>a</sub> *Provider* riceve il JSON<sub>a</sub>; grazie ai dati ricevuti è in grado di fornire il servizio richiesto dal *Client* nello *step* 1.

La modellazione dell'architettura e struttura del sistema da sviluppare seguirà questo prototipo di UC<sub>a</sub><sup>8</sup>. Il modello associato al caso asincrono con *callback* seguirà la stessa struttura e *step* dello UC<sub>a</sub> illustrato qui sopra, con l'aggiunta speculare del messaggio di ritorno.

---

<sup>1</sup>Simple Object Access Protocol

<sup>2</sup>Web Service Description Language

<sup>3</sup>eXtensible Markup Language

<sup>4</sup>XML Schema Definition

<sup>5</sup>JavaScript Object Notation

<sup>6</sup>Web Service

<sup>7</sup>REpresentational State Transfer

<sup>8</sup>Use Case

### 3.3 Progettazione architetturale

#### 3.3.1 Un *middleware* basato su un $EDA_a$

La progettazione architetturale ha portato alla produzione di diversi diagrammi  $UML_a$  per rappresentare efficacemente l'architettura del prodotto e fornire un modello da seguire durante il processo di programmazione. Il processo ha richiesto frequenti *meeting* e confronti per raggiungere un risultato finale soddisfacente al fine della sperimentazione.

La progettazione architetturale del prodotto comprende un *Middleware* centrale basato su di una *Event Driven Architecture* con l'utilizzo di Apache Kafka, e due componenti di *testing* chiamati *service client* e *service provider*, che simulano due servizi esterni che comunicano con il middleware attraverso  $REST_a$  *request*.

Il sistema è composto da molteplici microservizi, che comunicano attraverso la rete di Docker. L'obiettivo della progettazione è modellare un *Middleware* che sia implementabile in una *Service Oriented Architecture\_g*, e consentirne la verifica e collaudo grazie ai servizi di *testing*.

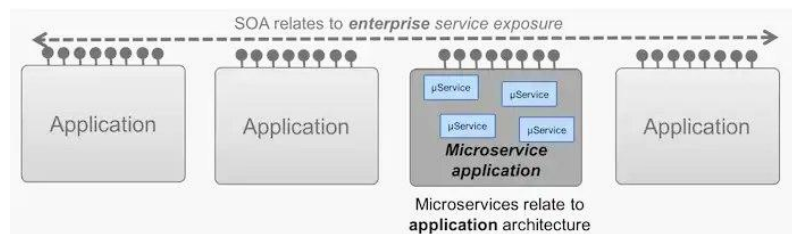


Figura 3.2: Visione ad alto livello delle differenze tra  $SOA_a$  e microservizi

<https://www.ibm.com/cloud/blog/soa-vs-microservices>

In figura 3.2 è rappresentata una visione ad alto livello dell'implementazione di un'applicazione a microservizi all'interno di una  $SOA_a$ , evidenziando la differenza tra i due concetti.

Per rappresentare in modo chiaro ed elegante il modello descritto dal  $UC_a$  sopra, ho elaborato diversi diagrammi  $UML_a$ . Questi diagrammi rappresentano i componenti *color coded*, notazione utilizzata per dare continuità e chiarezza attraverso le diverse tipologie di  $UML_a$  *diagrams*.

#### 3.3.2 $UML_a$ *sequence diagrams*

A partire dallo  $UC_a$  descritto nella sezione precedente, ho prodotto un  $UML_a$  *sequence diagram* più approfondito per rappresentare il flusso del  $JSON_a$  tra i vari componenti.

La progettazione del sistema associato al caso asincrono con *callback* comprende lo stesso schema  $UML_a$  del caso asincrono, con aggiunta del flusso di ritorno speculare flusso di andata.

La re-ingegnerizzazione del flusso sincrono è stata scartata in favore dello studio di funzionalità aggiuntive tramite l'utilizzo della piattaforma di *event streaming*. La progettazione di un sistema basato su questo flusso era inizialmente prevista (come requisito desiderabile) nel piano di lavoro iniziale poiché associata ad un caso d'uso reale (di cui si è parlato nelle sezioni precedenti), ma infine è stata giudicata poco opportuna e fuori dagli scopi di Apache Kafka, un sistema basato sull'asincronismo.

Il tempo associato a tale requisito è stato pertanto riproposto per testare un'altra funzione utile in un *Middleware* quali la trasformazione di alcuni dati presenti nel  $JSON_a$ . Più precisamente, è

stato aggiunto un dato sensibile che viene nascosto e sostituito con asterischi "\*" dopo la produzione del *topic* in Kafka grazie all'utilizzo di Kafka Streams.

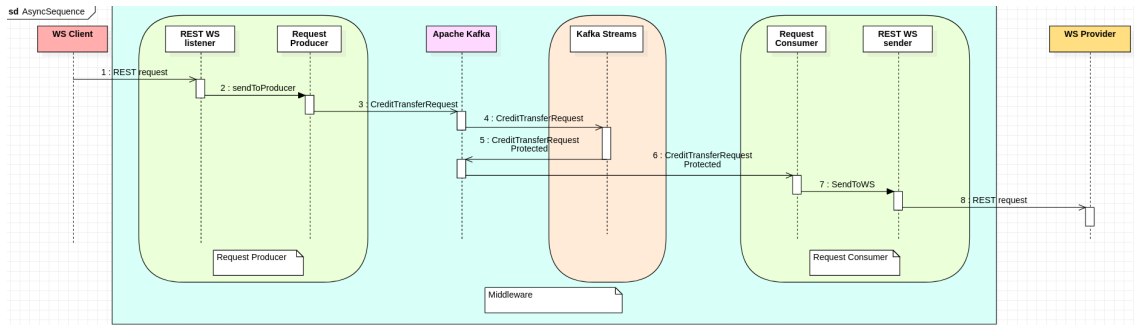


Figura 3.3: Diagramma di sequenza UML<sub>a</sub> per la re-ingegnerizzazione del flusso asincrono con protezione dei dati sensibili.

Fonte: elaborazione personale

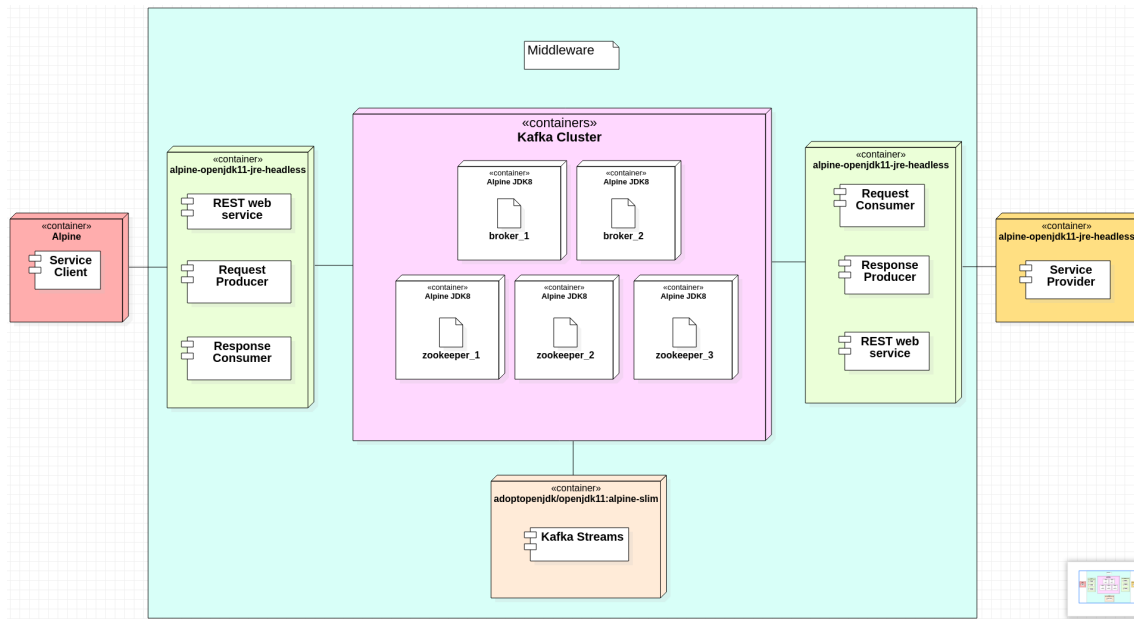
In figura 3.3 possiamo vedere il nuovo flusso asincrono con protezione (mascheramento) del dato sensibile. La versione con *callback* del flusso qui sopra prevede anche il flusso di ritorno (*callback*), non illustrato in figura ma facilmente intuibile grazie al grafico che la precede.

### 3.3.3 UML<sub>a</sub> deployment diagrams

A supporto di questi UML<sub>a</sub> *sequence diagram* che rappresentano efficacemente il flusso di dati, punto focale dell'intero sistema di integrazione (asincrono con la protezione del dato sensibile), ho prodotto ulteriori diagrammi, tra i quali il *deployment diagram*. Questo diagramma ha lo scopo di rappresentare la configurazione dei processi *run time*, modellando la struttura di base in cui eseguono i diversi servizi; il diagramma esprime l'ambiente in cui i vari componenti risiedono e ove essi comunicano tra di loro.

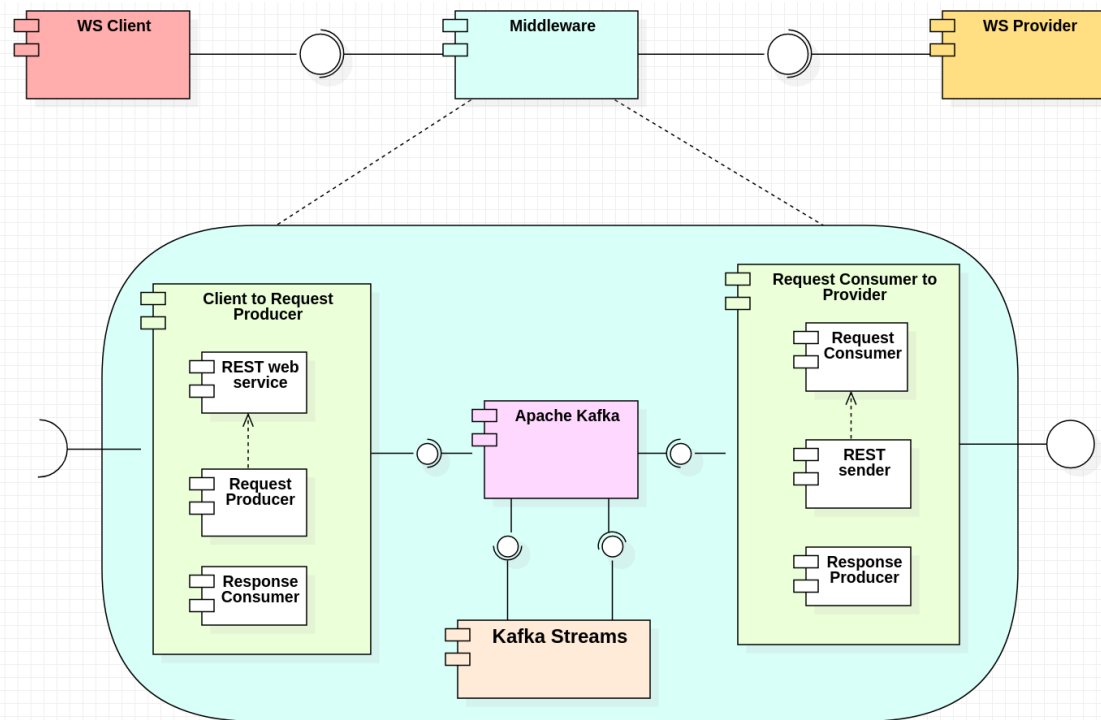
Con l'approvazione degli esperti aziendali, ho deciso di appoggiare il sistema di integrazione sulla piattaforma Docker.

Il diagramma di *deployment* (figura 3.4) vede pertanto l'utilizzo di numerosi *container* indipendenti che dialogano attraverso una rete locale all'interno di Docker. Questi *container* sono raffigurati dai vari nodi (rappresentati dai cubi in rilievo in figura). A questa notazione fa eccezione il nodo virtuale intitolato "Kafka Cluster", che ha solamente lo scopo di raggruppare i vari nodi legati all'*environment* di Kafka con funzione comune, ma che in realtà non compone un container reale a se stante. All'interno di questi nodi sono rappresentati gli artefatti che eseguono nel relativo *container*, per esplicitare la presenza dei componenti. Si può inoltre notare che l'ambiente di Apache Kafka è composto da un *cluster* composto da due servizi *Broker* e tre servizi *Zookeeper*, allo scopo di simulare un caso d'uso reale in cui i diversi componenti sono distribuiti in sistemi indipendenti e garantiscono l'affidabilità dello *streaming* di eventi.

Figura 3.4: UML<sub>a</sub> deployment diagram per la re-ingegnerizzazione del flusso asincrono (protetto)

Fonte: elaborazione personale

### 3.3.4 UML<sub>a</sub> component diagram

Figura 3.5: UML<sub>a</sub> component diagram per la re-ingegnerizzazione del flusso asincrono (protetto)

Fonte: elaborazione personale



Allo scopo di riassumere elegantemente i vari componenti del sistema ho elaborato un UML<sub>a</sub> *component diagram*. Esso riassume con una visione ad alto livello i componenti che compongono il sistema e le modalità con cui esse interagiscono attraverso le relative interfacce.

## 3.4 Programmazione

### 3.4.1 Kafka *cluster*

Il primo passo del processo di programmazione è stato quello del *setup* dell'ambiente di Kafka in locale. La guida rapida online fornita direttamente da Apache per un'installazione minimale di Kafka porta al *download* di un pacchetto e l'esecuzione di un servizio Zookeeper e un *broker* di Kafka.

Zookeeper è un servizio attualmente essenziale al funzionamento di Kafka, gestisce i nodi del *cluster* e mantiene una lista dei *topic* e dei messaggi. Le versioni future di Kafka renderanno questo servizio non necessario, ma attualmente sono ancora in fase di sviluppo e non adatta all'ambiente di produzione. I due *broker* si occupano di ricevere i messaggi dai *producer*.

Dopo un breve collaudo del corretto funzionamento dei servizi tramite l'inserimento di un evento in un *topic* e la relativa lettura, ho iniziato ad espandere e trasportare il sistema su dei *container* Docker.

Il *cluster* di Kafka utilizzato nel progetto di *stage* è composto dai componenti descritti dalla progettazione architetturale nella sezione precedente, ovvero tre servizi di Zookeeper e due *broker* per rappresentare un sistema di piccole dimensioni che tuttavia si avvicina ad un caso d'uso reale poiché contiene multipli Zookeeper e multipli *broker*.

Questi cinque servizi necessitano dunque di eseguire contemporaneamente in un ambiente "containerizzato" con Docker connessi alla stessa *network* locale, che consente ai diversi servizi di scambiare messaggi tra di essi.

Ho implementato questo *cluster* tramite un *file* di tipo *yml* utilizzato dall'estensione Docker-compose.

Il *file* contiene una lista di servizi, in cui in ognuno viene specificato:

- l'immagine Docker su cui viene costruito il servizio o il *dockerfile* da utilizzare per la sua costruzione;
- il nome del *container*;
- il nome del servizio;
- le dipendenze funzionali del servizio;
- l'indirizzo IPv4<sub>a</sub> statico e le porte attraverso cui è possibile raggiungere il servizio all'interno della rete locale dagli altri *container*.

### 3.4.2 Kafka *producer* e *consumer*

Il passo successivo è stato quello di sviluppare dei semplici *producer* e *consumer* in grado di inserire e leggere i dati in Kafka. Una volta collaudati rapidamente questi due eseguibili scritti in Java,

li ho incapsulati all'interno di un'applicazione utilizzando lo strumento di *build* e gestione delle dipendenze Maven<sup>9</sup>.

Per completare il modulo che costituisce il prototipo di *Middleware* è necessario che il *producer* e *consumer* siano in grado di comunicare con dei servizi esterni e fornire le interfacce adatte al ruolo. Nel caso più complesso del flusso asincrono con *callback*, entrambi questi servizi devono essere in grado di restare in ascolto di eventuali *REST<sub>a</sub> request* (che nel mio progetto utilizzano il protocollo HTTP<sub>a</sub>) e al tempo stesso di inviarle.

Ho pertanto creato un HTTP<sub>a</sub> *server* minimale in Java attraverso il *framework* Netty<sup>10</sup>, in esecuzione in un *thread* Java. Sviluppi futuri vedrebbero probabilmente l'utilizzo di un *framework* più evoluto come Spring Boot<sup>11</sup>. Un altro *thread* si occupa invece dell'invio delle *REST<sub>a</sub> request* al servizio di destinazione, una volta ricevuto un evento da parte del *consumer*.

L'insieme di questi componenti formano i blocchi logici illustrati in verde nella sezione precedente, in cui il primo servizio è formato da:

- *REST<sub>a</sub> Web Service* (HTTP<sub>a</sub> *server* in ascolto);
- *request producer*;
- *response consumer (callback)*;

e il secondo da:

- *request consumer*;
- *REST<sub>a</sub> Web Service* (invio della *request*);
- *response producer (callback)*.

### 3.4.3 WS<sub>a</sub> *Client* e WS<sub>a</sub> *Provider*

Al fine di testare il prototipo di *Middleware* prodotto, ho realizzato due ulteriori *container* con all'interno due eseguibili che si occupano di interagire con esso.

Il servizio intitolato WS<sub>a</sub> *Provider* è fondamentalmente simile ai servizi descritti nella sottosezione precedente: anch'esso possiede un HTTP<sub>a</sub> *server* per rimanere in ascolto delle *REST<sub>a</sub> request* inviate dal *request consumer* oltre ad un metodo per elaborare la *REST<sub>a</sub> request* di risposta associata al *callback*.

L'altro servizio, che idealmente potrebbe essere molto simile al precedente servizio di test, in realtà presenta delle considerevoli differenze. La causa di ciò non è una differenza reale in termini di funzionalità, quanto una questione di rapidità di *testing* e programmazione.

Il servizio è molto più leggero in termini di memoria, essendo composto da un semplice *container* con una distribuzione di Alpine Linux<sup>12</sup>; esso possiede un'installazione del *software* *curl* (accessibile via CLI<sub>a</sub>), ma è privo di ulteriori *software* aggiuntivi da me prodotti.

Gli obiettivi di questo WS<sub>a</sub> *Client* sono legati a due *network utility*: *curl* e *netcat*. La prima mi permette di eseguire, interagendo manualmente con il *container* tramite l'interfaccia CLI<sub>a</sub>, di eseguire *REST<sub>a</sub> request* con il JSON<sub>a</sub> di partenza. La seconda mi consente di restare in ascolto di eventuali *request* su di una porta a mia scelta.

<sup>9</sup>Maven - Introduction. URL: <https://maven.apache.org/what-is-maven.html>.

<sup>10</sup>Netty: Home. URL: <https://netty.io>.

<sup>11</sup>Spring Boot. URL: <https://spring.io/projects/spring-boot>.

<sup>12</sup>index | Alpine Linux. URL: <https://www.alpinelinux.org>.

### 3.4.4 Protezione dei dati sensibili con Kafka Streams

Come visto nella sezione 3.3, alcuni dei dati trasmessi nel JSON<sub>a</sub> vengono trasformati per mascherare i dati sensibili.

Precisamente, il JSON<sub>a</sub> passa dall'avere questa forma

```
1  {
2    "CallerSystem": "Sistema chiamante 1",
3    "PhoneNumber": "012345679",
4    "Currency": "EUR",
5    "Amount": "5",
6    "Info": "Causale del Trasferimento Credito Residuo",
7    "DebitType": "Bancomat",
8    "CreditTransferDate": "2021-07-21",
9    "CreditCardNumber": "1234567890123456"
10 }
```

Figura 3.6: JSON<sub>a</sub> inviato al *Middleware*

Fonte: elaborazione personale

a questa

```
1  {
2    "CallerSystem": "Sistema chiamante 1",
3    "PhoneNumber": "012345679",
4    "Currency": "EUR",
5    "Amount": "5",
6    "Info": "Causale del Trasferimento Credito Residuo",
7    "DebitType": "Bancomat",
8    "CreditTransferDate": "2021-07-21",
9    "CreditCardNumber": "*****"
10 }
```

Figura 3.7: JSON<sub>a</sub> protetto, ricevuto al termine del *callback*

Fonte: elaborazione personale

in cui si può notare che l'ultimo dato ha subito la modifica descritta.

Ho implementato questa funzione utilizzando Kafka Streams<sup>13</sup>, che permette di leggere un *topic* e modificarlo istantaneamente per un'elaborazione in *real time*.

### 3.4.5 Efficienza nello sviluppo

Durante la programmazione, ho adottato diverse misure per minimizzare il tempo necessario. Questi provvedimenti riguardano principalmente l'ottimizzazione del sistema di *container*, non allo scopo di migliorare l'efficienza, rapidità d'esecuzione e di *build* del prodotto finale (che è stata comunque raggiunta come effetto secondario) ma a quello di ridurre il tempo e risorse necessarie per il *testing*, riducendo di conseguenza le ore e le risorse necessarie allo sviluppo.

Tutte le misure prese sono strettamente legate alla mia familiarità con alcune tecnologie e al risorse personali richieste per l'apprendimento e sviluppo di una nuova funzionalità.

Elenco i principali provvedimenti intrapresi per minimizzare il tempo di sviluppo:

- **ottimizzazione delle risorse utilizzati dai *container*.** Molti dei container prodotti possiedono delle versioni *premade* sulla libreria di DockerHub<sup>14</sup> (ad esempio i *container* che

<sup>13</sup> *Kafka Streams: Introduction*. URL: <https://kafka.apache.org/28/documentation/streams/>.

<sup>14</sup> *Docker Hub Container Image Library | App Containerization*. URL: <https://hub.docker.com>.

formano il *cluster* di Kafka). Tuttavia, utilizzare delle versioni *ad-hoc* da me costruite mi ha permesso di ridurre considerevolmente le dimensioni del *container* e mantenere solamente le funzioni a me necessarie, e conseguentemente ridurre il tempo della *build* delle immagini e la loro esecuzione. Dato le numerose operazioni di *build* e *testing*, nel lungo termine ha portato ad un risparmio di tempo significativo.

- **invio della richiesta  $REST_a$  del *service client* tramite  $CLI_a$ .** I vantaggi di questa modalità manuale risiedono come anticipato nella rapidità di sviluppo, non solo di questo servizio ma anche dei restanti. Infatti grazie alle funzioni offerte da docker-compose, è semplice eseguire il *restart* di un singolo *container* contenente un servizio per testarne la nuova versione, e successivamente è sufficiente eseguire manualmente la *request* dal  $WS_a Client$  per verificare il funzionamento del sistema. È sicuramente possibile l'automatizzazione di questo processo (ad esempio effettuando richieste continue in modo automatico) ma l'implementazione di questa funzione avrebbe, secondo la mia stima personale, richiesto più tempo della soluzione attuata o posto problemi nel filtro dell'*output*.
- **la *build* delle applicazioni costruite con Maven avviene in locale.** Questo porta a due vantaggi. Il primo è che la *build* impiega meno tempo, dato che non è necessario lanciare alcun *container* e Maven non deve sincronizzare o controllare la versione delle dipendenze online più di volte (è possibile disattivare la sincronizzazione, ma richiede comunque più tempo). La seconda, più importante, è che l'immagine Docker stessa non richiede né *build* né Maven. Il *container* si occupa solamente di copiare l'eseguibile pre-costruito al suo interno e di eseguirlo con Java.

### 3.5 Collaudo del prodotto finale

Il prodotto finale del progetto è composto da i diversi componenti illustrati nella sottosezione 3.3.4, per un totale di dieci servizi ognuno nel proprio *container* Docker (conformi con il *deployment diagram* alla sottosezione 3.3.3).

Nella figura seguente è presente una bozza riassuntiva dalla *board* di progetto ClickUp.

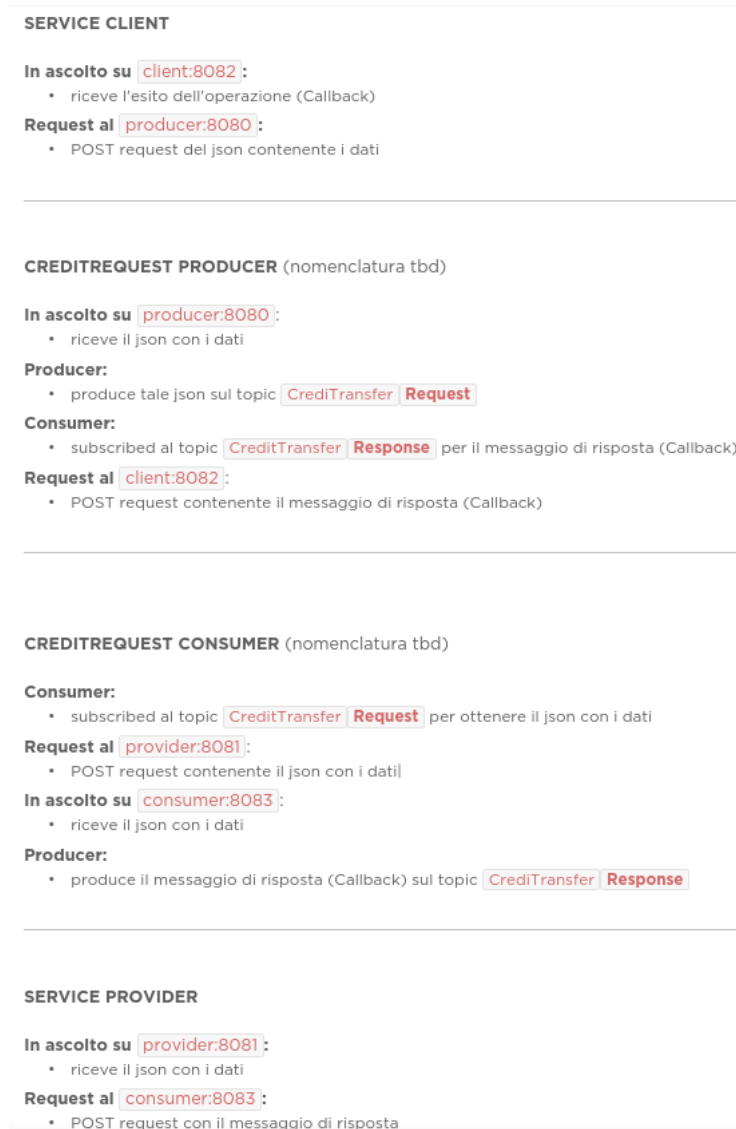


Figura 3.8: UML<sub>a</sub> Riassunto dei componenti prodotti nel caso asincrono con *callback* e della loro comunicazione

Fonte: elaborazione personale

L'immagine illustra i servizi prodotti per il flusso asincrono con *callback* e di come essi comunichino tra di loro attraverso le varie porte, elaborata durante il processo di Progettazione iniziale (in cui non è ancora presente la protezione del dato sensibile con Kafka Streams).

Una visione ad alto livello può esprimere il prodotto risultante dallo *stage* come **due servizi di test** quali **WS<sub>a</sub> Client** e **WS<sub>a</sub> Provider** che **scambiano messaggi sotto forma di file JSON<sub>a</sub> tramite un Middleware** basato su **Apache Kafka** in un'architettura focalizzata **sugli eventi**. Il *Middleware* presenta inoltre la funzionalità aggiuntiva di trasformazione dei dati grazie all'utilizzo di Kafka Streams.

A seguito di tutte le dovute verifiche del *software*, il prodotto è stato più volte collaudato con la supervisione del tutor aziendale e il responsabile del EAI<sub>a</sub>. Tutti i *container* prodotti sono stati revisionati, compresa la loro corretta esecuzione.

Il tempo dedicato al processo di Collaudo è stato relativamente breve rispetto alla maggior parte

dei prodotti *software*: la causa di questa durata minore è la natura sperimentale del progetto di *stage* che, come visto nella sotto-sezione 2.3.1, è incentrato sulla re-ingegnerizzazione del flusso di dati ma senza l'obiettivo di produrre un *Middleware* da implementare in produzione (quest'ultimo è un obiettivo aziendale nel lungo termine, ma non di questo specifico progetto di sperimentazione).

Questi collaudi hanno visto il processo di *build* e successivo avvio simultaneo di ogni singolo servizio all'interno del *cluster* tramite l'apposito comando di Docker-compose "**docker-compose up -build**". Dopodiché, al fine di testare che il *software* eseguisse effettivamente le funzioni richieste dal progetto di sperimentazione (*test* d'efficacia), ho effettuato delle HTTP<sub>a</sub> REST<sub>a</sub> *request* contenenti il JSON<sub>a</sub> con i dati. Ho eseguito questo processo connettendomi al *container* contenente il servizio WS<sub>a</sub> *client*, per poi dare manualmente il comando di *request* grazie all'*utility* **curl**. L'analisi del JSON<sub>a</sub> in entrata e in uscita da ogni servizio mi ha garantito l'efficacia del *software* sperimentale prodotto.

Infine, il prodotto finale è stato presentato all'azienda in conclusione del percorso di *stage* in un *online meeting* con l'aiuto di alcune diapositive da me prodotte e una dimostrazione *live* del *software* in esecuzione tramite la condivisione dello schermo.

# Acronimi

**API** Application Programming Interface. 12, 15, 16

**CLI** Command Line Interface. 25, 33, 35

**COBRA** Common Object Request Broker Architecture. 6

**EAI** *Enterprise Application Integration<sub>g</sub>*. 1, 4, 7, 9, 11, 12, 16, 17, 20, 21, 23, 25, 27, 28, 36

**EDA** Event Driven Architecture. 2, 4, 16, 18, 21, 29

**EJB** Enterprise Java Bean. 6

**ESB** Enterprise Service Bus. 1, 4, 9, 11

**HTTP** HyperText Transfer Protocol. 9, 33, 37

**ICT** Information and Communication Technologies. 5, 20

**IPv4** Internet Protocol version 4. 32

**J2EE** Java 2 Platform Enterprise Edition. 6

**JMS** Java Message Service. 6

**JSON** JavaScript Object Notation. 4, 9, 28, 29, 33, 34, 36, 37

**MOM** Message Oriented Middleware. 6, 10

**O-O** Object Oriented. 6

**OS** Operating System. 13

**P2P** Point To Point. 4, 11, 15, 17, 18

**REST** REpresentational State Transfer. 16, 25, 28, 29, 33, 35, 37

**SOA** *Service Oriented Architecture<sub>g</sub>*. 1, 4, 9, 11, 25, 28, 29

**SOAP** Simple Object Access Protocol. 9, 28

**SSO** Single Sign On. 12

**TCP** Transmission Control Protocol. 16

**UC** Use Case. 28, 29

**UML** Unified Modeling Language. 2, 4, 6, 29–32, 36

**VM** Virtual Machine. 4, 13, 14

**WS** Web Service. 2, 28, 33, 35–37

**WSDL** Web Service Description Language. 28

**XML** eXtensible Markup Language. 28

**XSD** XML Schema Definition. 28



# Glossario

## ***Enterprise Application Integration***

Il termine si riferisce al processo d'integrazione tra diversi tipi di sistemi informatici di un'azienda attraverso l'utilizzo di software e soluzioni architetturali.

Fonte: [https://it.wikipedia.org/wiki/Enterprise\\_Application\\_Integration](https://it.wikipedia.org/wiki/Enterprise_Application_Integration) . 11, 17, 21, 27, 38

## ***Service Oriented Architecture***

La Service Oriented Architecture definisce un modo per rendere i componenti software riutilizzabili tramite interfacce di servizio. Queste interfacce utilizzano standard di comunicazione comuni in modo da poter essere rapidamente integrate in nuove applicazioni senza dover eseguire ogni volta una profonda integrazione.

Ogni servizio in una SOA incorpora il codice e le integrazioni dei dati necessari per eseguire una funzione aziendale completa e discreta (ad esempio, il controllo del credito del cliente, il calcolo di un pagamento di un prestito mensile o l'elaborazione di un'applicazione ipotecaria). Le interfacce di servizio forniscono un accoppiamento libero, il che significa che possono essere richiamate con poca o nessuna conoscenza della sottostante modalità di implementazione dell'integrazione.

I servizi sono esposti utilizzando protocolli di rete standard - come SOAP (simple object access protocol)/HTTP o JSON/HTTP - per inviare richieste di lettura o modifica dei dati. I servizi sono pubblicati per consentire agli sviluppatori di trovarli rapidamente e riutilizzarli per assemblare nuove applicazioni.

Fonte: <https://www.ibm.com/it-it/cloud/learn/soa> . 1, 9, 11, 25, 28, 29, 38

# Bibliografia

- [1] *ClickUp™ / One app to replace them all.* URL: <https://clickup.com>.
- [2] *Cos'è il Middleware?* URL: <https://www.redhat.com/it/topics/middleware/what-is-middleware>.
- [3] *Coursera / Build Skills with Online Courses.* URL: <https://www.coursera.org>.
- [4] *Docker Hub Container Image Library / App Containerization.* URL: <https://hub.docker.com>.
- [5] *Empowering App Development for Developers / Docker.* URL: <https://www.docker.com/>.
- [6] *Google Chat.* URL: <https://chat.google.com>.
- [7] *index / Alpine Linux.* URL: <https://www.alpinelinux.org>.
- [8] *Kafka Streams: Introduction.* URL: <https://kafka.apache.org/28/documentation/streams/>.
- [9] *Maven - Introduction.* URL: <https://maven.apache.org/what-is-maven.html>.
- [10] *Netty: Home.* URL: <https://netty.io>.
- [11] *Notion - The all-in-one workspace for your notes, tasks, wikis, and databases.* URL: <https://www.notion.os>.
- [12] *Online Courses - Learn Anything, On Your Schedule / Udemy.* URL: <https://www.udemy.com>.
- [13] *Oracle VM Virtual Box.* URL: <https://www.virtualbox.org/>.
- [14] *Overview of Docker Compose / Docker Documentation.* URL: <https://docs.docker.com/compose/>.
- [15] *Spring Boot.* URL: <https://spring.io/projects/spring-boot>.
- [16] *Sync Lab.* URL: <https://www.synclab.it/>.