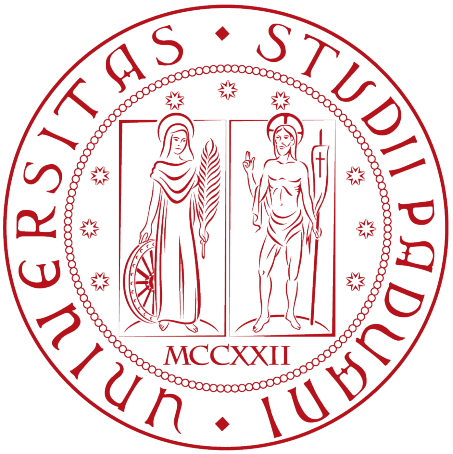
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Un confronto tra Java Spring e Node.js per il back-end di web app

Tesi di laurea triennale

Relatore

Prof.Tullio Vardanega

Laureando

Giacomo Bulbarelli

Anno Accademico 2022-2023

Sommario

Il presente documento descrive il lavoro che ho svolto durante il periodo di *stage*. L'attività è stata da me affrontata periodo dal 5 Settembre 2022 al 30 Ottobre 2022, con una durata effettiva di 300 ore, presso l'azienda SyncLab S.p.A ed è stata moderata dal mio *tutor* aziendale, l'ingegnere Pallaro Fabio, e dal relatore interno all'Università, il Professor Vardanega Tullio.
Lo scopo dello *stage* era di studiare e confrontare due differenti tecnologie nell'ambito dello sviluppo di applicativi *web* utilizzando l'architettura a microservizi, dandomi l'occasione di valutarne aspetti positivi e negativi.
L'attività è stata dunque divisa in tre momenti distinti, ciascuno volto a un obiettivo differente:

1. Studio di natura teorica delle tecnologie utilizzate;
2. Produzione di un applicativo che facesse utilizzo di suddette tecnologie, esponendone le funzionalità più rilevanti;
3. Discussione delle due tecnologie, sotto forma di analisi comparativa, rispetto alle implementazioni prodotte.

Ho strutturato il contenuto di questo elaborato in 4 capitoli:

- **Capitolo 1**: presentazione dell'azienda, dal punto di vista organizzativo, tecnologico e rispetto agli ambiti in cui opera;
- **Capitolo 2**: presentazione dell'offerta di *stage*, con enfasi sul valore che tale progetto ha avuto per l'azienda e per la mia persona;
- **Capitolo 3**: presentazione dettagliata del progetto, con una disamina dei concetti teorici alla base delle tecnologie utilizzate, delle componenti fondamentali che le compongono e dell'applicativo realizzato;
- **Capitolo 4**: bilancio ragionato dei risultati ottenuti, della maturazione professionale a postieriori del tirocinio e delle competenze fornite dal corso di studi in relazione al tirocinio.

Presenterò quanto detto corredando il testo con immagini, provvedendo a riportarne la fonte contestualmente. Nel caso in cui questa non sia presente, per convenzione, verrà sottinteso che l'immagine è stata prodotta da me.

"It's not binary. You can be decent and gifted at the same time."

— Steve Wozniak

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Tullio Vardanega, relatore della mia tesi, per l'aiuto e il sostegno fornitomi durante la stesura dell'elaborato.

Ringrazio i miei compagni di università Giuseppe, Veronica, Gianmarco, Mariano, Enrico, Emanuele e Alessandro per aver condiviso con me le gioie e le difficoltà di questi anni.

Ringrazio i miei amici Fabio, Francesca, Alberto, Andrea, Lorenzo e Damiano, compagni di gioco e di vita.

Ringrazio Andrea e Giulia, insostituibili amici di sempre.

Ringrazio Marco, Linda, Emma, Cinzia e Gaetano per aver condiviso con me momenti preziosi e per essere stati porto sicuro nella tempesta.

Ringrazio Anita per aver attraversato con me la tempesta, per essere stata al mio fianco nonostante tutto e per aver sempre creduto che io potessi farcela.

Ringrazio Massimiliano, Maria Luisa, Camilla, Giovanni ed Emanuele, la mia famiglia, per avermi sostenuto e aiutato sempre in questi anni.

Padova, Febbraio 2023 Giacomo Bulbarelli

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei codici sorgenti.

Capitolo 1

L'azienda: synclab s.r.l.

1.1 L'azienda

L'azienda presso cui ho svolto lo *stage* è SyncLab S.r.L, nata nel 2002 a Napoli e attiva nel settore dell'*Information and Communication Technology* (spesso abbreviato con la sigla ICT).
Nel corso degli anni, grazie anche ad una forte spinta verso la ricerca, si è espansa aprendo nuove sedi, aumentando il numero di collaboratori e evolvendo i servizi che offre, passando dalla sola produzione *in-house* di soluzioni per i propri clienti all'integrazione e evoluzione di infrastrutture pre-esistenti.
Per dare una migliore contestualizzazione alle affermazioni precedenti, riporto di seguito i dati forniti direttamente dall'azienda.

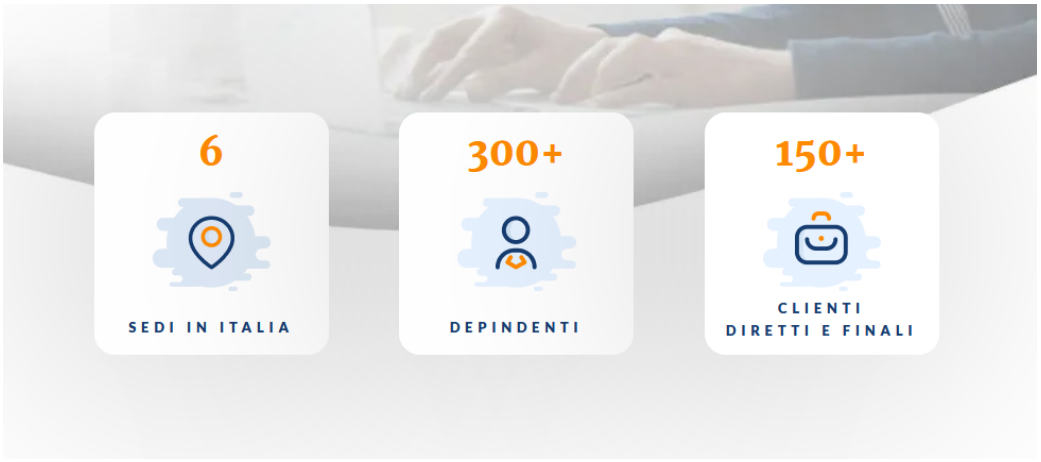


Figura 1.1: Dati relativi all'azienda: sedi presenti sul territorio italiano, numero di dipendenti assunti e numero dei clienti con cui collabora.
Fonte: synclab.it

Le sedi si trovano nelle località di Napoli, Roma, Milano, Verona, Padova e Como.

A ciascuna di queste fa riferimento un numero variabile di dipendenti ma, grazie ad una struttura interna che cerca di favorire la collaborazione, l'interazione non è limitata ai colleghi della singola sede e diventa globale a livello aziendale. Tale scelta è da ricondursi alla volontà di promuovere il *knowledge sharing* interno all'azienda e di creare un ambiente in cui il miglioramento personale non è frutto dei soli sforzi individuali ma anche di una collaborazione attiva tra i singoli. Nella mia personale esperienza ho avuto modo di interagire con le sedi di Verona (presso cui si è effettivamente svolto lo stage) e di Padova. Ritengo sia inoltre molto importante sottolineare come, dal punto di vista anagrafico, i dipendenti nei contesti da me osservati siano tutti molto giovani.

1.2 Settori di attività e *partner*

SyncLab S.r.L., come già accennato, opera principalmente nel settore dell'*Information and Communication Technology*.
Sebbene ad inizio attività fosse classificabile come *Software house* nel corso degli anni si è affermata anche e soprattutto come *System integrator*.
La differenza tra i due ambiti risulta quindi essere particolarmente rilevante per comprendere meglio il *modus operandi* dell'azienda. Con il termine *Software house* si identifica un azienda che, avendo identificato un *opportunity* nel mercato, sviluppa internamente delle soluzioni software che la soddisfino e offre i propri prodotti ai clienti interessati. D'altro canto con il termine *System integrator* si identifica invece un azienda che, contattata da aziende esterne, effettua manutenzione e evoluzione delle funzionalità di prodotti software già sviluppati e in uso.
Si tratta quindi di due approcci allo sviluppo molto diversi: mentre in un caso il rapporto con il cliente avviene spesso a valle dello sviluppo dell'applicativo, nel secondo esso è parallelo e funzionale alla realizzazione del prodotto stesso.
Ritengo sia bene sottolineare come quanto detto finora si concentri principalmente sulle modalità operative dell'azienda, ma un ragionamento realmente utile a inquadrarne i settori di attività non può non considerare quali siano i domini applicativi con cui si confronta. SyncLab S.r.L. collabora con numerosi clienti che operano in contesti differenti. In termini operativi, questo si traduce in un impegno da parte dell'azienda in ambiti variegati: dall'*E-commerce* e l'*Internet banking* alla logistica, la *cyber-security* e l'*EHealth*.

D'altro canto, essendo nata come *Software house* e quindi come azienda dedita alla produzione interna di software, conta numerosi progetti all'attivo, quali:

- **SynClinic**: *software* sviluppato nell'ambito dell'*EHealth* per favorire la gestione integrata di tutti i processi clinici e amministrativi di ospedali, cliniche e case di cura. Pensata per fornire supporto al personale nella gestione del rischio clinico, permette di gestire, organizzare e monitorare tutte la fasi del percorso di cura del paziente;

- **Fast Reservation**: *software* sviluppato nell'ambito *web* per la gestione delle prenotazioni per ogni tipo di pubblico esercizio, come ad esempio spiagge, aree verdi e ogni attività che opera nel settore food;

- **Clean Roads**: *software* sviluppato nell'ambito *web* per consentire il *tracking* dei sistemi di pulizia stradale sia in termini di posizione geografica (attraverso il sistema GPS) sia in termini di dati raccolti dai sensori con cui ciascuna unità viene equipaggiata;

- **Sobereye**: *software* sviluppato nell'ambito *web* per monitorare facilmente lo stato psico-fisico di un individuo, attraverso l'analisi della pupilla. Tali controlli sono pensati per rilevare alterazioni dovute ad affaticamento o consumo di alcool e droghe, al fine di limitare gli incidenti sul posto di lavoro.

Di seguito riporto un istantanea delle informazioni disponibili direttamente sul sito aziendale che illustra alcuni degli ambiti in cui l'azienda opera.

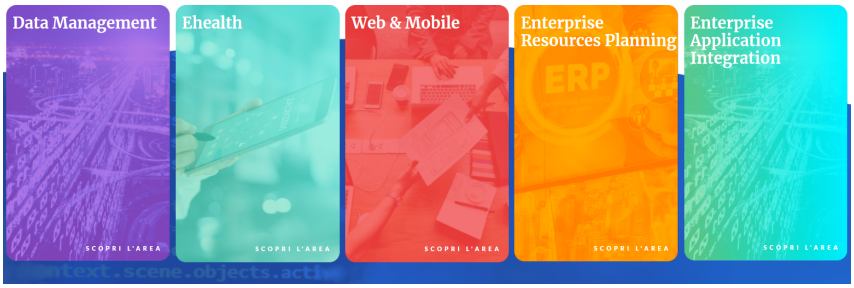


Figura 1.2: Alcuni degli ambiti in cui l'azienda opera. Si può osservare direttamente dal sito aziendale come vi sia varietà nei domini applicativi.
Fonte: synclab.it

Oltre a quanto mostrato nella figura precedente, grazie al sito aziendale ho potuto anche constatare l'identità di alcune delle aziende con cui collabora: Trenitalia, Rai, PosteItaliane, Unicredit, Intesa San Paolo, Sky, Eni, Enel, Fastweb e Vodafone.

1.3 Tecnologie interne utilizzate

Internamente l'azienda si avvale di un gran numero di tecnologie differenti per gestire i diversi aspetti della produzione *software*. In questa sezione tratterò le tecnologie della quale ho avuto un esperienza diretta senza però, per ovvie ragioni, scendere nel dettaglio sul come queste vengano internamente utilizzate per la gestione del lavoro. Esporrò, nell'ordine, prima i linguaggi con cui l'azienda implementa i propri prodotti, seguiti dai *framework* impiegati per strutturare suddette implementazioni e dagli strumenti di comunicazione e coordinamento dei lavori.
È subito interessante osservare come l'azienda si avvalga di diversi linguaggi di programmazione differenti, ciascuno applicato al contesto più adatto:

- **Java**: linguaggio di programmazione orientato agli oggetti estremamente diffuso. Grazie anche alla sua grande portabilità, tra i vari ambiti in cui viene utilizzato spicca la creazione di *back-end* per *web-app*;

- **Python**: linguaggio di programmazione multi paradigma, noto per la sua apparente semplicità e la sua grande diffusione in numerosi ambiti tra cui il *Machine Learning* e lo sviluppo *web*;

- **Javascript**: linguaggio di programmazione multi paradigma nato originariamente con lo scopo di rendere dinamiche le risorse *web*. Nel corso degli anni, grazie

all'evento di tecnologie come il *runtime* Node.js, ha trovato grande applicazione nello sviluppo di applicativi e, in particolare, di risorse *web* e applicativi *mobile*. Questa "conversione" nel dominio applicativo del linguaggio, non prevista al momento della sua creazione, ha portato alla luce diverse criticità nel suo utilizzo e alla successiva creazione di una versione migliorativa, ovvero Typescript;

- **Typescript**: linguaggio di programmazione multi paradigma nato come estensione del linguaggio *Javascript* allo scopo di colmarne numerose lacune e mancanze. Utilizzato principalmente nell'ambito della creazione di risorse *web* e di applicazioni *mobile*, negli ultimi anni ha ricevuto grande supporto in termini di adozione e sviluppo da parte di aziende e sviluppatori;

- **PL/SQL**: linguaggio di programmazione procedurale, creato come estensione del linguaggio *SQL*. Viene principalmente utilizzato per interagire con *database* relazionali, permettendo di scrivere *query* SQL senza la necessità di utilizzare un linguaggio che funga da intermediario per la definizione di procedure complesse.

I linguaggi sopra esposti vengono raramente utilizzati facendo unicamente affidamento alle funzionalità che questi espongono *out of the box*, e spesso sono accompagnati da *framework* che rendono lo sviluppo più rapido e comprensibile anche a chi non possiede competenze approfondite nel contesto specifico.
Questi si rendono necessari per dare una struttura (*framework* può infatti essere tradotto proprio con il termine *struttura*) all'implementazione del codice, facendo sì che il lavoro del singolo sia disciplinato al di là di quella che può essere l'interpretazione personale sul come implementare una specifica funzionalità. Nel caso in cui si renda necessario implementare nuove funzionalità le modalità da seguire sono normate dal *framework* scelto, e il lavoro del singolo consiste nel capire come fare quanto richiesto restando dentro le logiche imposte.
Alcuni esempi in questo senso sono:

- **Java Spring**: framework nato per agevolare lo sviluppo di applicativi *web* e *desktop* con il linguaggio Java, composto da diversi moduli ciascuno utile alla risoluzione di problematiche specifiche. Grazie a una struttura che promuove la creazione di architetture facilmente manutenibili, si è imposto come standard *de facto* per la creazione di servizi *web* in Java;

- **Angular**: *framework* per lo sviluppo di pagine *web* con il linguaggio Javascript. Nato dalla volontà di semplificare lo sviluppo e migliorare le prestazioni degli applicativi prodotti, si è saputo imporre negli anni come una delle soluzioni più utilizzate nel settore.

- **Odoo**: *framework* per lo sviluppo di *software* gestionali in Python. La sua struttura modulare permette di comporre soluzioni *ad hoc*, modellando le funzionalità esposte sulle specifiche esigenze dell'utente.

Per quanto si cerchi di normare e regolamentare il lavoro è però impensabile che un gruppo di individui possa sincronizzare le attività senza un adeguato supporto. Lo sviluppo mediante le tecnologie appena esposte è pertanto mediato da *software* che, oltre ad agevolare la codifica, permettono anche di sincronizzare e storicizzare quanto prodotto.
Nel contesto dello sviluppo *software* è tanto importante sapere e poter comunicare efficacemente i problemi incontrati e condividere le proprie conoscenze quanto lo è saper

condividere il proprio operato senza causare regressioni e malfunzionamenti critici del prodotto. Pertanto ritengo sia bene sottolineare che il termine *supporto* riferito a un *software* non riguarda il contesto della comunicazione con il *team* ma si concentra anche sul rendere agevole la manutenzione e la correzione degli applicativi.
In questo senso, alcuni esempi degni di nota sono programmi come:

- **Git**: *software* per il controllo versione (che spesso viene identificato con la sigla VCS, *Version Control System*. Con tale termine si identificano tutti quegli applicativi che permettono di tracciare le modifiche apportate al *software* in sviluppo andando a storicizzarle. Importante osservare inoltre che Git risulta essere un sistema di controllo versione distribuito, implicando quindi che ciascuno degli sviluppatori che contribuisce al progetto (oltre a condividere il proprio codice attraverso un *repository* condiviso) possiede una copia locale del codice sorgente, sulla quale opera in maniera indipendente prima di condividere il proprio operato;

- **VS Code**: editor di testo *language agnostic* (quindi non progettato per lo sviluppo con uno specifico linguaggio) che negli ultimi anni ha acquisito notevole popolarità per la sua natura *open-source* e per l'ampia disponibilità di moduli che integrano le funzionalità già disponibili *out of the box*;

- **IntelliJ IDEA**: *software* per lo sviluppo integrato (identificato per mezzo della sigla IDE, *Integrated Development Software*, mira a semplificare la codifica del *software* con il linguaggio Java attraverso funzionalità che supportino attivamente lo sviluppatore. Ciò può avvenire in modi diversi, passando dall'analisi statica del codice (utile per segnalare errori logici e sintattici prima che esso venga eseguito) all'integrazione con *software* esterno come il già citato Git per il versionamento del codice.

Per la comunicazione interna invece troviamo soluzioni come:

- **Google Meet**: *software* per videoconferenze, parte di un insieme più vasto di applicativi noto con il nome di Google Suite. Complice anche la situazione creatasi durante la pandemia di SARS-CoV-2, viene ad oggi ampiamente utilizzato e sebbene esistano soluzioni alternative la sua naturale integrazione con gli altri applicativi prodotti da Google lo rendono particolarmente appetibile;

- **Google Calendar**: *software* gestionale per la creazione di calendari privati e condivisi tra più utenti, parte di un insieme più vasto di applicativi noto con il nome di Google Suite. Viene utilizzato per sincronizzare gli impegni dei singoli dipendenti e organizzare l'alternanza tra *smart-working* e lavoro in presenza;

- **Trello**: *software* gestionale in stile Kanban che implementa il meccanismo di gestione del lavoro della Scrum *board* nel modello *agile*. Attraverso la creazione di schede apposite associate a *task* e la loro ripartizione in colonne definite sulla base dei possibili stati di avanzamento dei *task* stessi, permette di monitorare lo stato dei lavori e favorisce una migliore sincronizzazione tra i vari membri del team di sviluppo.

- **Discord**: *software* per la comunicazione vocale e testuale. Emulando il modello dei *software* di *chat* che sfruttano il protocollo *Internet Relay Communication* (spesso abbreviato con la sigla IRC), permette la creazione di canali testuali dedicati al fine di rendere la comunicazione *topic based* e estende questo concetto anche ai canali di *chat* vocale. È interessante osservare come, nonostante il

software non sia nato con lo scopo di agevolare la comunicazione in ambito aziendale, viene comunque adottato per la semplicità di utilizzo e per le numerose funzionalità proposte;

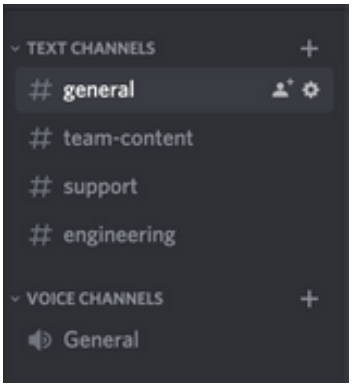


Figura 1.3: Screenshot del client Discord, che mostra una possibile ripartizione di canali testuali e vocali. È immediatamente riconoscibile la struttura tipica dei servizi IRC, dove un canale testuale viene rappresentato per mezzo del carattere # seguito dal nome del singolo canale.
Fonte: Google immagini, *keyword*: "Discord *screenshot*"

Finora ho presentato le tecnologie sotto forma di elenco, limitandomi a renderne esplicita la natura e il senso d'essere. Nella sezione successiva, analizzerò come queste possono cooperare per formare un prodotto *software* completo.

1.4 Dallo *stack* tecnologico ad applicativo

Le tecnologie sopra elencate, anche se esposte individualmente, non sono utilizzate "atomicamente" ma in quanto parte di uno *stack* tecnologico.
Per dare concretezza a quanto appena detto, ritengo sia utile portare un esempio concreto facendo riferimento in particolare all'ambito degli applicativi *web*. Essi vengono generalmente divisi in due macro-aree di sviluppo: *Front-end*, che si occupa di realizzare componenti software destinate all'interazione con l'utenza dell'applicativo finale, e *Back-end*, che si occupa invece di definire le procedure di manipolazione dei dati a disposizione del sistema e garantirne la persistenza per una corretta fruizione. Ciascuno di questi due ambiti richiede che venga selezionato uno *stack* tecnologico adeguato. Per la realizzazione di un interfaccia *web*, riprendendo dalla *pool* di tecnologie elencate in precedenza, si potrebbe quindi proporre l'adozione del *framework* Angular (progettato e realizzato proprio a tale scopo), che dal canto suo richiede necessariamente l'utilizzo di tecnologie quali (tra le molte coinvolte) il linguaggio di programmazione Typescript e del linguaggio di *markup* HTML5. Si tratta quindi di tecnologie diverse tra loro per natura, ma che funzionano in maniera coesa e costituiscono di fatto lo *stack* citato in precedenza.

È interessante osservare come la scelta di un *framework*, in questo caso, vincoli **necessariamente** anche le altre tecnologie utilizzate. Ciò non è sempre vero, ed

è possibile osservarlo spostandosi dal contesto del *Front-end* a quello relativo allo sviluppo *Back-end*.
Riprendendo sempre dalle tecnologie esposte in precedenza, si potrebbe decidere di utilizzare il *framework* Java Spring per la realizzazione delle logiche di *processing* dei dati e di comunicazione unito ad un qualunque DBMS *(Database Managment System)* per gestire la persistenza dei dati. Rispetto a prima, sebbene la scelta del linguaggio ava sia obbligata, quella invece relativa al DMBS **non è** vincolata dalla scelta del *framework*.

A questi due insiemi di tecnologie ne viene spesso affiancato un terzo, che include tutti gli applicativi che facilitano le attività di codifica, di versionamento e di comunicazione all'interno di un team. In questo caso la scelta sugli applicativi da utilizzare dipende dal *way of working* definito per il progetto.
Come fatto finora, quindi riprendendo dalle tecnologie presentate in precedenza, si può ipotizzare uno *stack* tecnologico anche per questo contesto, prendendo ad esempio un IDE come IntelliJ IDEA per lo sviluppo con il linguaggio Java e un editor di testo come VSCode per lo sviluppo con il linguaggio Typescript, un *software* di versionamento come Git e applicativi quali Discord e Trello per la comunicazione interna e la gestione delle attività di progetto.

Gli insiemi di tecnologie appena visti (gli *stack* di cui ho parlato in precedenza) vengono poi messi in relazione tra loro, andando a formare l'insieme di tecnologie che nella sua totalità può essere impiegato in un qualunque progetto.
Di seguito riporto una figura riassuntiva di quanto detto finora, dove vengono raggruppate a livello visivo (per ambito di pertinenza) le tecnologie esposte nella sezione precedente.

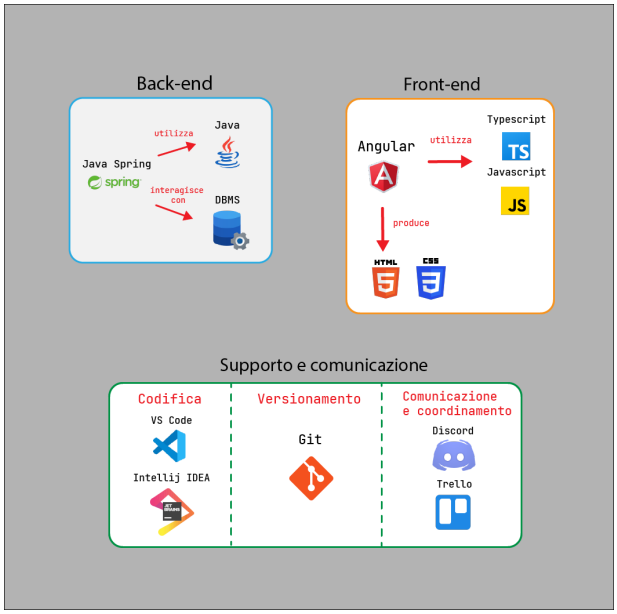


Figura 1.4: Esempio di *stack* tecnologico per una *web-app*.

È quindi fondamentale non considerare le tecnologie che vengono utilizzate inter-

namente da SyncLab (e da qualunque team di sviluppo) in maniera individuale, ma come parte di una visione più ampia e ad alto livello, che pone l'accento sul "cosa" si vuole fare piuttosto che sul "come".
È tuttavia necessario, a valle dei ragionamenti appena fatti, fare alcune precisazioni: in primo luogo, gli *stack* proposti **non** sono da intendere come unica possibile scelta, ma sono stati esposti in modo da riprendere le tecnologie utilizzate internamente da SyncLab ed esemplificare dei possibili scenari. In secondo luogo, ma non meno importante, gli *stack* sono stati ipotizzati partendo dalla scelta di un *framework* e andando a vincolare quindi le altre tecnologie necessarie per la realizzazione della componente specifica. Questo non è l'unico criterio utilizzabile e non è mia volontà in nessun modo cercare di spingere il lettore verso una scelta che segue il medesimo raziocinio. Le tecnologie possono essere scelte sulla base di molti fattori differenti e la trattazione dei criteri di scelta non è prevista in questa sede.

1.5 Propensione all'innovazione tecnologica

La propensione per l'innovazione tecnologica rappresenta per l'azienda uno degli aspetti più importanti, e viene applicata su più livelli. Ciò è possibile a partire dalla scelta che la stessa opera in termini di organizzazione del lavoro e metodologie di lavoro. Internamente infatti viene abbracciato il modello di sviluppo *agile*, e in particolare la metodologia *Scrum*. Questo modello di sviluppo è stato scelto perchè rende lo sviluppo reattivo a eventuali cambiamenti dei requisiti, non vincolandoli e permettendo (attraverso un coinvolgimento attivo dei committenti) di produrre applicativi di reale valore. Di seguito riporto un immagine di una visione ad alto livello della metodologia appena descritta.

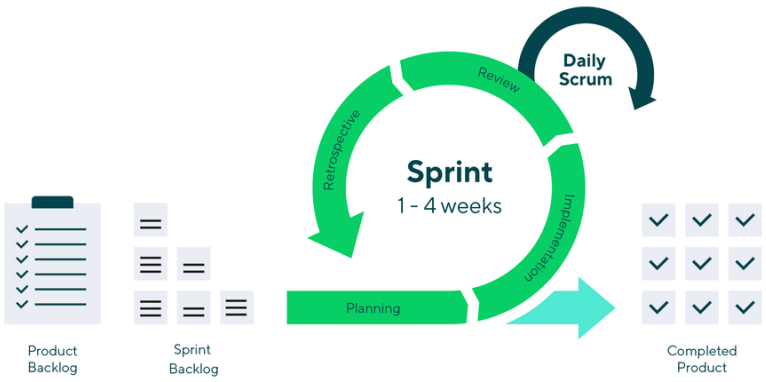


Figura 1.5: La metodologia *agile* Scrum.

Calando la metodologia nel contesto aziendale, questa si articola nella maniera seguente:

1. Definizione di un *product backlog* che include un elenco completo dei *task* da svolgere. Questi verrano inseriti in una *board* utile a tenere traccia del loro stato di avanzamento;

2. Definizione di uno *sprint backlog* che include un elenco completo dei *task* che dovranno essere portati a termine durante uno *sprint*;

3. Pianificazione dello *sprint*, ovvero del periodo di durata variabile durante cui verrano svolti i *task* definiti nel *sprint backlog*;

4. Esecuzione delle attività di sviluppo, il vero e proprio *sprint*, di durata variabile;

5. Revisione dello *sprint*, per mezzo della valutazione retrospettiva di quanto prodotto, utile a raffinare lo sviluppo e garantire incrementi significativi.

Tale metodologia è stata applicata anche all'attività di *stage* da me affrontata, a riprova del fatto che l'azienda la abbraccia pienamente. Ovviamente l'azienda cerca di mantenere un alto profilo anche attraverso l'adozione di nuove tecnologie per lo sviluppo di prodotti *software*, sia per quanto concerne l'implementazione di funzionlità sia per quanto riguarda il coordinamento dei lavori e la moderazione del progetto.
Le modalità con cui l'azienda si manitene aggiornata in questo senso sono molteplici:

- Internamente l'azienda dispone di un *team* di R&D,*(Research and Development)* grazie alla quale può sperimentare le nuove tecnologie attraverso la creazione di applicativi all'avanguardia da proporre al mercato di riferimento;

- Le attività di tirocinio che l'azienda istanzia in affiliazione agli enti di formazione sono spesso incentrate sulla creazione di applicativi che possono fungere da spunto per lo sviluppo di applicazioni utili nel contesto culturale contemporaneo o sull'analisi di tecnologie innovative ancora sconosciute internamente;

- Internamente l'azienda promuove e sostiene l'autoformazione che i dipendenti affrontano durante il loro percorso di lavoro, fornendo l'accesso a corsi di formazione specifica su *topic* di rilievo e stimolando il *knowledge sharing* tra i singoli dipendenti.

Capitolo 2

Il progetto: analisi comparativa tra Spring e NestJS

2.1 Rapporto dell'azienda con le attività di tirocinio

All'interno del contesto aziendale le attività di tirocinio sono gestite mantenendo in primo luogo molto attivi i rapporti che l'azienda ha con enti come scuole di formazione e università attraverso tutto il territorio italiano. Nella figura presentata di seguito è possibile osservare, tramite informazioni fornite direttamente dall'azienda stessa, alcuni dei citati enti con cui l'azienda collabora.

Figura 2.1: Alcuni degli enti con cui l'azienda collabora
Fonte: synclab.it

La stretta collaborazione dell'azienda con enti di formazione è proprio uno dei principali fattori che permette di proporre attività che risultano essere di valore e per l'azienda stessa e per l'individuo coinvolto nelle attività. Secondo la mia esperienza infatti suddette attività sono finalizzate all'esplorazione e l'approfondimento di tecnologie e tecniche implementative innovative, ignote all'azienda al momento dello svolgersi dell'attività ma spesso materia di corsi o esami presso gli enti che fungono da bacino per i potenziali candidati. L'obiettivo è quindi arricchire la *knowledge base* interna, permettendo di esplorare nuove tecnologie senza dover impiegare organico interno e risorse di managment. Allo stesso tempo, ciò risulta essere vantaggioso anche dal punto di vista economico, considerando che le spese necessarie ad affrontare le attività di tirocinio (principalmente di carattere assicurativo nei confronti del soggetto che affronta l'attività) sono quasi totalmente coperte dall'ente con cui viene instaurato il rapporto collaborativo. L'individuo che affronta l'attività è quindi avvicinato ad attività che gli permettono di arricchire il proprio bagaglio di conoscenze, e grazie a proposte che vengono progettate *ad hoc* sulla base del percorso da lui seguito e dai suoi interessi, l'azienda riesce a rendersi competitiva ed appetibile anche agli occhi dei soggetti che decide di coinvolgere, senza per questo dover rinunciare ai vantaggi già citati in precedenza. Nonostante come detto vi sia varietà tra le attività proposte (soprattutto in termini di contenuti dal punto di vista tecnologico) queste possono essere fondamentalmente ricondotte a tre tipologie specifiche, ciascuna con finalità differenti:

1. **Evoluzione di *software* pre-esistente**: l'azienda offre allo stagista la possibilità di evolvere le funzionalità proposte per un applicativo già sviluppato. Tali funzionalità sono spesso atomiche, non vengono utilizzate per la produzione finale del *software* ma rappresentano potenziali soluzioni alternative che l'azienda tiene in considerazione durante lo sviluppo;

2. **Studio teorico di nuove tecnologie**: l'azienda propone allo stagista di studiare dal punto di vista teorico nuove tecnologie oppure di approfondire aspetti teorici specifici di tecnologie già note e utilizzate internamente;

3. **Studio applicativo di nuove tecnologie**: l'azienda propone allo stagista di studiare dal punto di vista applicativo nuove tecnologie oppure di approfondire aspetti operativi specifici di tecnologie già note e utilizzate internamente.

La seconda e terza proposta possono essere istanziate contemporaneamente (come ad esempio nel mio caso), affiancando studio teorico e applicazione della tecnologia in un unico percorso.
Questo tipo di approccio, unito a quanto già discusso nella sezione 1.4, permette all'azienda di mantenere aggiornate le competenze e le conoscenze rispetto alle tecnologie utilizzate ma anche di controllare attivamente quale sia il grado di maturità di tecnologie di interesse e che desidera adottare.
Infine, ma non meno importante, le esperienze di *stage* sono utilizzate anche come banco di prova per eventuali futuri dipendenti: durante il periodo di *stage* l'azienda valuta attentamente l'operato del tirocinante e nel caso in cui il rapporto risulti essere mutuamente benefico viene offerta una posizione interna. Questo tipo di approccio risulta essere particolarmente efficace: i dipendenti con cui ho avuto modo di confrontarmi hanno in più occasioni sottolineato come l'aver seguito questo percorso li abbia aiutati a inserirsi e integrarsi agilmente con l'ambiente aziendale, sia prima che dopo l'esperienza di tirocinio.

Calandoci nel contesto specifico dell'attività di tirocinio a me proposta, questa mi è stata presentata durante l'evento *StageIT*, organizzato dall'Università di Padova per permettere agli studenti che devono sostenere tale impegno di entrare in contatto con le aziende e muovere i primi passi in questa nuova tipologia di rapporto con anche il sostegno dell'ente Universitario.
Sebbene l'edizione a cui ho partecipato (quella svoltasi nell'anno solare 2021) fosse ancora condizionata dalle problematiche sorte in concomitanza con la pandemia da SaRS-COV-2 e si sia svolta in modalità puramente telematica, ho avuto comunque modo di confrontarmi con diverse realtà aziendali di rilievo nel panorama italiano, quali Zucchetti, San Marco Informatica e SyncLab.
Trovo sia degno di nota osservare inoltre che tali aziende erano già a me note prima dell'evento stesso, trattandosi di realtà che collaborano con la facoltà di Informatica dell'Università di Padova anche al di fuori del contesto delle attività di tirocinio (ad esempio, nel mio caso specifico, le aziende appena citate collaborano con il corso di Ingegneria del Software nel proporre i capitolati per le attività di progetto). Ciò, unito al (naturale ma non evidente) mio essere totalmente alieno alla produzione *software* in ambito aziendale, mi ha introdotto ai diversi contesti permettendomi di studiarli anzi tempo.
Come intuibile rispetto a quanto già esposto nel capitolo 1 alla fine la mia scelta è ricaduta su *SyncLab s.r.l.* e i motivi legati a tale scelta sono esposti ed esaminati nelle sezioni successive.

2.2 Progetto proposto

L'attività a me proposta nasce a partire da quanto detto nella sezione "Propensione all'innovazione tecnologica", quindi dalla volontà dell'azienda di aggiornare le tecnologie utilizzate internamente partendo dagli sviluppi che il panorama contemporaneo offre.
Nel corso degli anni, e in particolare dall'avvento del *runtime* Node.js, nell'ambito dell sviluppo *web* si è potuto osservare un grande fermento grazie alla possibilità di uniformare i linguaggi utilizzati per lo sviluppo del *back-end* e il *front-end* di applicativi. Grazie proprio a *Node.js* è possibile implementare le funzionalità che governano il *back-end* di un applicativo *web* utilizzando il linguaggio *Javascript*, che viene contestualmente impiegato anche per l'implementazione di funzionalità nell'ambito del *front-end*. Passando più concretamente all'attività, questa consiste in un analisi comparativa tra due *framework* per lo sviluppo di *back-end* in ambito web. Il primo di questi, Java Spring (di cui si è già data una breve introduzione nella sezione 3 del capitolo 1) è considerato standard *de facto* nel settore, e presenta soluzioni robuste e ben collaudate a molti problemi che possono insorgere durante lo sviluppo. Essendo però una tecnologia realizzata a inizio millennio molte di queste soluzioni sono state implementate a posteriori rispetto alla pubblicazione e per quanto efficaci hanno col tempo aumentato la complessità dello sviluppo in maniera significativa.
Si è quindi potuto osservare l'insorgere di nuove tecnologie, che partendo proprio da questo aumento di complessità cercano di migliorare quanto offerto da Spring riprendendone però anche i punti di forza, ed è in questa ottica che si propone NestJS.
NestJS è un *framework* che, avvalendosi di una tecnologia diversa (quella del *runtime* Node.js per il linguaggio Javascript), cerca quindi di snellire lo sviluppo dal punto di vista architetturale e implementativo senza però rinunciare ai progressi fatti in questo senso da Java Spring.
Ecco quindi che sorge spontanea la domanda centrale cui questa esperienza tenta di

dare una risposta:

Come si pone NestJS nel contesto dello sviluppo *web* se confrontata con le già mature (ma, a tratti, convolute) soluzioni proposte da Java Spring?

La trattazione di questo argomento viene poi calata all'interno di un altro tema di grande rilievo nel panorama attuale, ovvero i **microservizi** e il loro sviluppo. Questo, unito a quanto detto in precedenza, ci porta all'esperienza offerta concretamente dall'azienda e ai prodotti attesi a posteriori:

Studio dell'architettura a microservizi e dei *framework* in esame e sviluppo di microservizi atomici che forniscono le medesime funzionalità per mezzo delle singole tecnologie.

L'immagine che segue tenta di mostrare quanto detto in precedenza anche dal punto di vista grafico, evidenziando come il fulcro dell'idea di progetto sia l'implementazione di microservizi sfruttando due tecnologie distinte e differenti, ma implementando le medesime funzionalità.

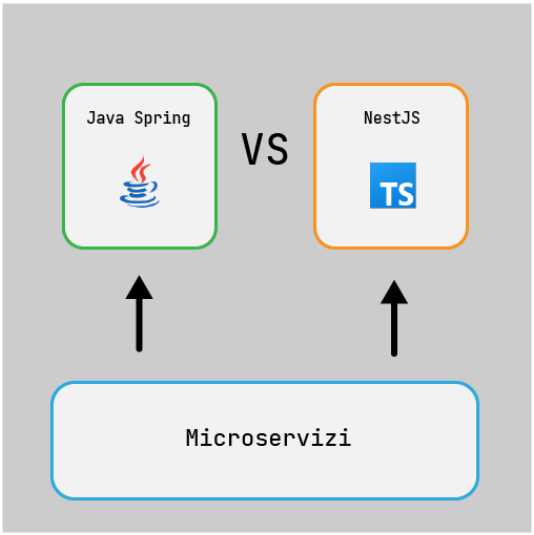


Figura 2.2: Una rappresentazione visiva del raziocinio alla base dell'attività proposta. Si vuole evidenziare il percorso incrementale in termini di acquisizione di competenze nello sviluppo web, partendo dall'architettura a microservizi e applicandola a due *framework* differenti

Come già brevemente accennato nella sezione *Rapporto dell'azienda con le attività di tirocinio*, le proposte di tirocinio non sono cieche rispetto al contesto aziendale, e rappresentano progetti che fattivamente hanno un valore per l'azienda stessa. Nel caso dell'attività a me proposta, in accordo a quanto spiegatomi durante il mio percorso dal mio referente interno, questa è propedeutica a una futura ed eventuale adozione del *framework NestJS* nell'ambiente di produzione. Sebbene siano disponibili un gran numero di alternative in questo senso, quella a me proposta risulta essere particolarmente

appetibile per l'azienda a causa della sua vicinanza (dal punto di vista dei meccanismi che ne regolano il funzionamento e lo sviluppo) ad un altra tecnologia già utilizzata internamente ovvero *Angular* (per lo sviluppo del *front-end* di *web app*).
Osservando quindi le tecnologie coinvolte e quanto detto in precedenza si può capire quindi quale sia il reale valore che tale offerta può portare al contesto aziendale: partendo da una tecnologia già nota e utilizzata per lo sviluppo di *back-end* di *web app* (*Java Spring*) studiarne un'altra *NestJS* che implementa le stesse soluzioni attraverso un tecnologia differente pur non risultando totalmente aliena grazie al suo abbracciare concetti già propri di altre tecnologie che sono note e utilizzate all'interno del contesto aziendale *Angular*. L'obiettivo è implementare i medesimi *microservizi* atomici sfruttando le tecnologie citate in precedenza, definendo i medesimi casi di test per entrambi gli applicativi. Essendo l'attività finalizzata ad un analisi comparativa tra le due tecnologie e il *way of working* che implementano, i microservizi attesi come prodotto finale non sono studiati per risolvere un problema specifico ma fungono da vetrina per le funzionalità rilevanti offerte dai *framework*. Questa idea, anche se apparentemente banale, evidenzia immediatamente un aspetto fondamentale del prodotto finale. È infatti impensabile nel mondo della produzione *software* che un applicativo risolva un problema adattando una tecnologia specifica al problema stesso, ma al contrario la progettazione di qualunque soluzione deve partire dal problema e scegliere la tecnologia che lo risolve cercando di massimizzare efficacia ed efficienza.
È proprio in quest'ottica che il mio referente interno ha modellato lo scenario che il servizio tenta di implementare, ovvero la gestione di conti finanziari (tralasciando eventuali dettagli di stampo economico) per una generica utenza. In questo senso, i microservizi da implementare (per mezzo e di Java Spring e di NestJS) sono:

- Gestione della generica utenza;
- Gestione di portafogli/conti legato a un utente specifico;
- Gestione delle transazioni tra due utenti.

Ciascuno di essi, come detto, dovrà essere implementato due volte avendo cura di replicare le medesime logiche quanto più accuratamente possibile (compatibilmente con i limiti sintattici imposti dal linguaggio). In questo modo, una volta terminati i prodotti, l'analisi potrà effettivamente concentrarsi sulle tecnologie utilizzate e sul modo in cui queste abilitano (o meno) specifiche funzionalità di interesse.

2.3 Organizzazione del tirocinio

Il progetto di tirocinio si è svolto dal 5 Settembre 2022 al 30 Ottobre del medesimo anno, per una durata complessiva di 8 settimane. Durante il corso del periodo di tirocinio l'azienda ha predisposto che il lavoro venisse svolto in presenza (presso la sede di Verona) almeno una volta a settimana (non vincolandomi però a presentarmi in sede non più di una volta a settimana). Per la moderazione del lavoro durante la durata del tirocinio l'azienda ha strutturato un piano di lavoro, seguendo le direttive fornite dall'Università degli studi di Padova in materia, al fine di garantire il corretto svolgimento dei lavori e definire quanto più precisamente possibile le attività da svolgere. Nello specifico, il piano di lavoro è strutturato nella maniera seguente:

- Prima settimana (40 ore):

  - Incontro con persone coinvolte nel progetto per discutere i requisiti e le richieste relativamente al sistema da sviluppare;
  - Verifica credenziali e strumenti di lavoro assegnati;
  - Ripasso Java Standard Edition e *tool* di sviluppo (IDE, ecc.);
  - Studio teorico dell'architettura a microservizi: passaggio da monolite a microservizi con pro e contro;
  - Studio teorico dell'architettura a microservizi: Api *Gateway*, *Service Discovery* e *Service Registry*, *Circuit Breaker* e *Saga Pattern*.

- Seconda settimana (40 ore):
  - Studio Spring Core / Spring Boot;
  - Studio servizi REST e *framework* Spring Data REST.

- Terza settimana (40 ore):
  - Studio ORM, in particolare *framework* Spring Data JPA.

- Quarta settimana (40 ore):
  - Ripasso Javascript;
  - Studio Node.js e test di implementazioni *back-end*.

- Quinta settimana (40 ore):
  - Studio dei *framework* Express.js e NestJS;
  - Studio delle funzionalità da implementare e scrittura casi di *test*.

- Sesta settimana (40 ore):
  - Implementazione dei servizi in Spring Framework.

- Settima settimana (40 ore):
  - Implementazione dei servizi in Node.js.

- Ottava settimana (40 ore)
  - Test, considerazioni e collaudi finali.

Naturalmente, per poter verificare il corretto svolgimento delle attività si è reso necessario definire delle modalità di comunicazione che fossero efficaci nel permettere di capire se i lavori si stessero svolgendo regolarmente o se invece fosse necessario intervenire in maniera correttiva. Durante lo svolgersi delle attività la comunicazione tra la mia persona e il mio referente interno volta alla rendicontazione del lavoro svolto si è strutturata dividendosi in 3 modalità distinte, che differiscono per frequenza e tipologia. Di seguito sono riportate suddette modalità, esposte secondo la frequenza con cui queste venivano istanziate, dalla meno frequente alla più frequente:

- **Colloquio settimanale**: un colloquio settimanale (in presenza o da remoto) durante il quale veniva discusso il lavoro svolto. Nel caso in cui le attività prevedessero uno studio puramente teorico, esponevo le conoscenze acquisite al mio referente interno, facendo riferimento ai materiali consultati e esponendo eventuali dubbi incontrati. Nel caso in cui le attività settimanali prevedessero

la codifica o la configurazione di applicativi *software* il colloquio prevedeva un primo momento di analisi del codice, seguito dalla una valutazione del lavoro svolto e un analisi critica rispetto a possibili soluzioni alternative ai problemi incontrati.

- ***Board* Trello**: con l'inizio delle attività il mio tutor interno ha predisposto una *board* (utilizzando la piattaforma Trello) all'interno della quale è stato riversato il cronoprogramma del piano di lavoro concordato. Ciascuna settimana è quindi stata convertita in una *card* specifica e le singole attività sono state associate alla settimana di riferimento. Per poter controllare lo stato di avanzamento dei lavori la *board* è stato poi divisa in 4 colonne, nominate secondo gli stadi successivi in cui il lavoro poteva trovarsi:
  - *Backlog*: insieme di attività da svolgere;
  - *In corso*: insieme di attività attive;
  - *In verifica*: insieme di attività da discutere e approvare durante il colloquio settimanale;
  - *Completate*: insieme di attività verificate e approvate.

  Le attività una volta avviate non potevano più tornare nella colonna *Backlog* ed dovevano obbligatoriamente essere terminate, mentre le attività presenti nella colonna *Completate* non potevano essere più spostate in colonne differenti.
  Questa modalità di rendicontazione del lavoro è stata istanziata per permettere "a colpo d'occhio" di intuire quale fosse lo stato di avanzamento del progetto di tirocinio nella sua totalità.

- **Foglio Google Sheet**: con l'inizio delle attività il mio tutor interno ha predisposto un foglio di calcolo sulla piattaforma Google Sheet per la rendicontazione giornaliera dei lavori. Tale foglio è stato diviso in 4 colonne. La prima colonna doveva da me essere compilata inserendo la data di riferimento per il lavoro svolto, mentre la seconda colonna doveva essere compilata sempre da parte mia inserendo il contenuto (brevemente riassunto) dei materiali studiati/prodotti durante la giornata. La terza colonna e la quarta colonna erano invece riservate (dal punto di vista compilativo) al mio tutor interno, ed erano adibite rispettivamente a evenutali note sul lavoro svolto e un *check-box* per segnalarmi l'avvenuta presa visione dei contenuti da me inseriti.

Quanto detto finora si applica alla sola interazione tra la mia persona e l'azienda, mentre invece la comunicazione e la rendicontazione del lavoro con il mio tutor interno universitario, il Professore Tullio Vardanega, sono state interamente normate secondo le istruzioni fornite direttamente dal Professore. Esse avvenivano attraverso la stesura di un breve resoconto testuale il cui contenuto consisteva in un riassunto delle attività svolte e una breve sezione in cui queste venivano messe in relazione alle attese del piano di lavoro. La redazione del documento avveniva su base settimanale, nello specifico ogni 5 giorni lavorativi effettivi.

Tutti questi meccanismi di moderazione erano propedeutici alla corretta realizzazione dei prodotti attesi al termine del tirocinio, che consisteva nella produzione di un applicativo tramite la tecnologia Node.js, lo sviluppo di un applicativo tramite la tecnologie Java Spring e un analisi comparativa tra le attività di sviluppo e le due tecnologie utilizzate.

In aggiunta a ciò sono stati fissati degli obiettivi che, una volta raggiunti, avrebbero decretato il termine delle attività e permesso di dare un giudizio complessivo sul lavoro svolto. Questi, in particolare, possono appartenere a 3 categorie distinte ovvero **Obbligatori**, **Desiderabili** e **Facoltativi**. L'appartenenza o meno di un obiettivo a una di queste categorie è dipesa sostanzialmente dal valore che (nella visione aziendale) il raggiungimento dell'obiettivo in questione era in grado di dare al tirocinio.

Nello specifico, i requisiti **Obbligatori** prevedono:

- Acqusizione di competenze sulle tematiche descritte nel piano di lavoro;
- Portare a termine l'implementazione dei microservizi con una percentuale di superamento pari al 80;
- Capacità di raggiungere gli obiettivi richiesti in autonomia seguendo il cronoprogramma;

Gli obiettivi **Desiderabili** consistevano in realtà di un unica voce, ovvero portare a termine l'implementazione dei microservizi con una percentuale di superamento pari al 100, e lo stesso ragionamento si applica anche agli obiettivi **Facoltativi**, il cui unico membro prevedeva uno studio su come poter realizzare le best practice dell'architettura a microservizi con Node.js dal punto di vista teorico. Per misurarne la qualità a valle dello sviluppo, l'azienda si aspetta di ricevere come prodotti *Software* il codice di ciascuno dei microservizi atomici la cui qualità è garantita dall'implementazione contestuale dei test di unità associati. I test in questione devono necessariamente avere una percentuale di superamento maggiore o uguale al 75. Trattandosi di un analisi comparativa che mira a valutare anche la qualità dell'esperienza di utilizzo dei due *framework*, oltre alla percentuale di superamento dei test verranno riportate, per essere confrontate, il numero di righe di codice implementate e il numero di file prodotti. Ciò verrà fatto per ciascuno degli applicativi prodotti, mettendo in relazione il numero dei file (sorgenti e di configurazione) e delle righe di codice alle metodologie che i due *framework* adottano nell'esporre le specifiche funzionalità.

2.4 Valore di progetto per la mia persona

Dal punto di vista del valore che il progetto proposto ha per la mia persona, questo deve essere considerato in quanto naturale continuazione di quello che è stato il mio percorso di attività di progetto all'interno dell'ambiente universitario. Durante il mio percorso di studi, infatti, mi sono trovato spesso ad affrontare attività di progetto che hanno richiesto l'acquisizione di competenze nell'ambito dello sviluppo web. Nello specifico:

- **Progetto per il corso di Basi di Dati**: attività progettuale che ha richiesto la realizzazione di una base di dati attraverso modello relazionale per la gestione di dati estrapolati da un contesto fittizio ma ben definito.
- **Progetto per il corso di Tecnologie Web**: attività progettuale che ha richiesto la modellazione e la realizzazione di un servizio *web* utilizzando, tra le altre tecnologie, il linguaggio Javascript.
- **Progetto per il corso di Ingegneria del Software**: attività progettuale che, nell'ambito della logistica automatizzata, ha richiesto la realizzazione di 3

componenti *software*: un interfaccia utente attraverso il framework Angular, una componente *software* che simulasse il comportamento di unità a guida autonoma sviluppata attraverso la tecnologia *Node.js* e un *server* che fosse in grado di elaborare i dati ricevuti e comunicare sia con l'interfaccia utente sia con il simulatore per unità a guida autonoma.

Queste attività, riportate secondo l'ordine cronologico con cui sono state affrontate, possono essere inquadrate nell'ottica di un percorso di esplorazione e studio delle tecnologie e delle metodologie proprie del settore dello sviluppo *web*.
Di seguito riporto una rappresentazione visiva del concetto appena esposto, ovvero il collocamento dell'attività di tirocinio in un percorso di maturazione rispetto alle competenze da me acquisite negli anni nell'ambito dello sviluppo *web*.

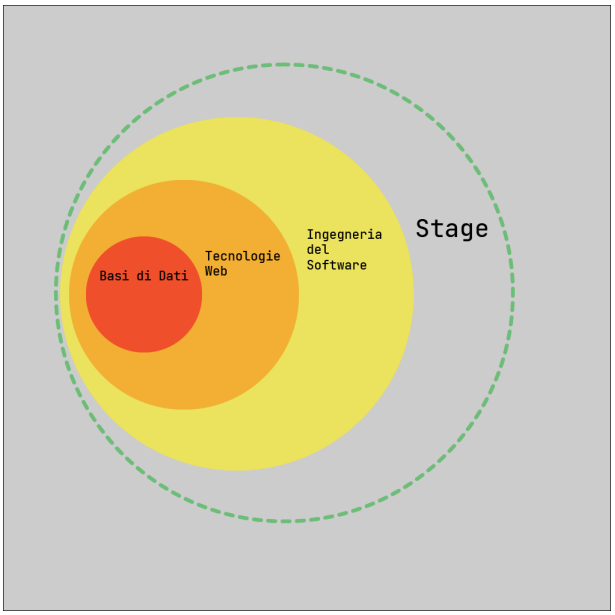


Figura 2.3: Una rappresentazione visiva del percorso esposto in precedenza.

L'attività proposta è risultata essere quindi appetibile in quanto:

- Naturale continuazione del mio percorso di apprendimento metodologico e tecnologico;
- Tratta di tematiche attuali e richiede un approfondimento specializzante di temi già noti;
- Associa le mie esperienze precedenti ad un contesto aziendale reale, stimolando e mettendo alla prova quelle che sono comunemente note come *soft-skills* (comunicazione con i colleghi, capacità di portare a termine task in autonomia e utilizzo critico di strumenti collaborativi).

In questo senso, due sono gli obiettivi che personalmente mi sono proposto di raggiungere durante il corso del progetto: in primo luogo, dare maggiore concretezza

al concetto di microservizio, spesso trattato dal punto di vista teorico senza però avere riferimenti concreti (implementativi), e successivamente raffinare e migliorare le mie competenze nell'implementazione di servizi REST.
In entrambi i casi la parola chiave è *approfondimento* e gli obiettivi proposti devono necessariamente essere considerati tenendo a mente quanto detto in precedenza, quindi come parte di un percorso coerente rispetto allo studio dei servizi *web*. Al fine di dimostrare la riuscita comprensione delle tematiche affrontate, l'obiettivo personale che mi sono posto è riuscire ad implementare i microservizi e verificarne l'efficacia attraverso l'implementazione dei rispettivi casi di test, esposti successivamente, con una percentuale di superamento maggiore o uguale a 90.

2.5 Considerazioni ulteriori

Nel prossimo capitolo verranno esposti i contenuti che ho affrontato durante l'attività di *stage*. Sono doverose e necessarie alcune premesse in merito proprio ai contenuti appena citati:

- Le nozioni che verranno da me esposte prima di parlare degli applicativi prodotti sono di carattere puramente teorico. Queste si concentrano principalmente su alcune componenti che normalmente vengono sviluppate nel contesto degli applicativi a microservizi. Nonostante queste vengano effettivamente trattate dal punto di vista teorico (e dal punto di vista aziendale sia stato valutato utile per me vedere questi contenuti), i concetti esposti sono stati trattati unicamente in via teorica e non è prevista una loro implementazione nel contesto di questa attività di tirocinio. Gli unici prodotti *software* attesi sono i microservizi atomici, privi quindi delle componenti architetturali e infrastrutturali necessarie a rendere l'applicativo completo e funzionante nella sua interezza.

- Un'altra importante precisazione riguarda la natura dell'applicativo prodotto. In un contesto di sviluppo aziendale un applicativo viene sviluppato **partendo** dal problema e delineando le tecnologie da impiegare sulla base proprio del problema da risolvere, nel contesto del mio tirocinio il *software* viene sviluppato funzionalmente nel tentativo di valutare come ciascuno dei due *framework* approccia la risoluzione dei medesimi task. Ciò che ritengo sia importante in questo caso è tenere a mente che gli oggetti della valutazione sono, in primo luogo, le tecnologie coinvolte e il modo in cui favoriscono lo sviluppo. I risultati ottenuti, soprattutto dal punto di vista delle funzionalità che gli applicativi offrono, devono essere visti nell'ottica in cui la valutazione non è tanto sulla qualità delle implementazioni fornite (che comunque deve essere verificata e validata) ma soprattutto su come ciascuna delle tecnologie affrontate approccia, favorisce, struttura e guida lo sviluppo.
  La qualità degli applicativi ottenuti deve pertanto essere calata nel contesto della tecnologia utilizzata, e deve concentrarsi principalmente su una valutazione del *way of working* proprio del *framework* in esame.

Capitolo 3

Il progetto: svolgimento

3.1 Servizi da implementare: funzionalità e architettura

Terminato l'inquadramento generale e lo studio delle componenti proprie dell'architettura a microservizi (contenuti riportati in appendice), insieme al *tutor* interno è stata delineata l'architettura dei singoli servizi da implementare come prodotto dell'attività di tirocinio e su cui si basano le considerazioni dell'analisi comparativa. Nella sezione 2 del capitolo 2 ho brevemente accennato al dominio applicativo fittizio sulla base della quale sono stati modellati i servizi, ovvero la gestione di utenti cui è associato un conto per la gestione di risorse finanziarie. Riprendendo sempre quanto detto nella già citata sezione, per ciascuno dei *framework* presi in esame l'obiettivo posto è quello di implementare 3 microservizi così ripartiti:

- Gestione della generica utenza;
- Gestione di portafogli/conti legato a un utente specifico;
- Gestione delle transazioni tra due utenti.

Per ciascuno di questi servizi il *pattern* architetturale applicato al modello a microservizi è quello della *Layered Architecture*. Ciò che ritengo essere importante sottolineare è che nel contesto di un architettura di questo tipo le dipendenze tra gli strati (i *layer* che si possono osservare in figura) vanno dallo strato più alto a quello più basso della gerarchia (sempre in riferimento alla figura presentata). Operativamente parlando, questo si traduce in funzioni (o classi) che richiamano altre funzioni (o metodi) che, logicamente, appartengono allo strato immediatamente sottostante.

Quanto appena detto può essere visivamente rappresentato come segue:

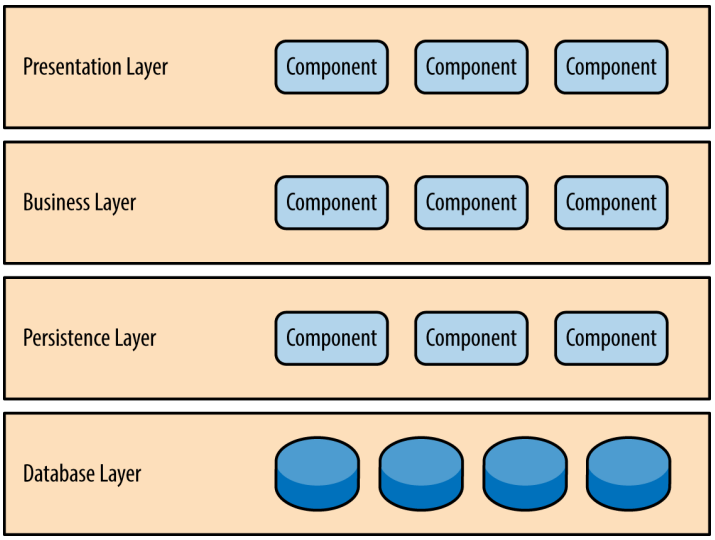


Figura 3.1: Rappresentazione visiva di una generica *Layered architecture*. Gli strati (il termine inglese *layer* può essere tradotto con la parola italiana strato) sono rappresentati da rettangoli, e ciascuno di essi può essere implementato per mezzo di una funzione o, nel contesto della programmazione orientata agli oggetti da una classe, che ne modella il comportamento.
Fonte: Google immagini, *keyword*: *Layered architecture diagram image*

Calandoci nel contesto degli applicativi da sviluppare, questi sono composti da 3 strati ciascuno, ognuno adibito ad un ruolo diverso. In particolare questi sono composti da:

- Uno strato adibito alla gestione delle richieste in ingresso;
- Uno strato che implementa le logiche di *business* del singolo servizio;
- Uno strato che gestisce la comunicazione tra l'applicativo e il database su cui viene fatta serializzazione e deserializzazione dei dati gesititi dal servizio.

Mi riferisco generalmente al primo degli strati appena descritti con il termine di ***Controller Layer***, al secondo di questi con il termine ***Service Layer*** e al terzo con il termine ***Repository Layer***. Tale nomenclatura acquisirà significato nelle sezioni successive, dove viene utilizzata contestualmente ai *framework* permettendo di capire meglio le motivazioni legate a tale scelta.

Passando invece alle funzionalità vere e proprie che i servizi implementano, queste sono state modellate a partire dai casi d'uso fittizi individuati con il proponente in sede di progettazione e riportati di seguito. La nomenclatura relativa ai singoli casi d'uso è la seguente:

$$UC[\alpha]$$

dove la componente α rappresenta l'identificativo del singolo caso d'uso ed è espressa come numero intero maggiore o uguale a 1.

Tabella 3.1: Tabella dei casi d'uso.

Identificativo	Caso d'uso
UC-1	Deve essere possibile registrare un nuovo utente fornendo una combinazione univoca di nome e email cui viene associato un portafogli.
UC-2	Deve essere possibile recuperare gli utenti registrati al servizio e i dati relativi al portafogli associato.
UC-3	Deve essere possibile eliminare un utente registrato al servizio e i dati relativi al portafogli associato.
UC-4	Deve essere possibile modificare le informazioni relative al nome e email di un utente registrato.
UC-5	Deve essere possibile effettuare transazioni tra gli utenti registrati al servizio.
UC-6	Deve essere possibile recuperare le informazioni legate allo storico delle transazioni associate ad un utente.

Il raziocinio alla base di questi non è quello di modellare un servizio efficace nel risolvere il problema di partenza (che è appunto fittizio e artificioso nella sua formulazione), ma di permettere in sede di implementazione di sfruttare alcune caratteristiche e funzionalità specifiche dei *framework* presi in esame. Questo concetto, a questo punto ribadito più volte, è fondamentale perché pone l'accento sul "come" e non sul "cosa" è stato prodotto.

3.2 Meccanismi e *pattern* trasversali ai *framework*

Prima di addentrarmi nella disamina relativa all'implementazione effettiva dei servizi, dedico questa sezione alla valutazione ed esposizione di alcuni meccanismi e *pattern* implementati da entrambi i *framework* permettendomi così nelle sezioni successive di concentrarmi su quella che è la struttura specifica in relazione alla tecnologia in esame.

La prima e più importante caratteristica di questi *framework* riguarda l'organizzazione architetturale dei progetti che è possibile creare, sviluppare e mantenere per mezzo di questi. In particolare, questa si struttura (come già accennato nella sezione precedente) seguendo il modello a strati (*Layered architecture*). Il primo di questi, denominato con il termine *Controller*, è adibito all'implementazione delle logiche di comunicazione per mezzo di API REST. Concretamente questo si traduce nell'implementazione di metodi che mappano una specificia tipologia di chiamata HTTP a una sequenza di istruzioni da eseguire, che nel contesto della programmazione ad oggetti si traduce nelle chiamate a metodi appartenenti ad altre classi. Tali classi implementano quindi le logiche di dominio, e dal punto di vista architetturale sono associate secondo degli strati proposti chiamato appunto *Service*. Le logiche implementate in questo caso non sono adibite alla comunicazione quindi ma alla manipolazione dei dati, che si svolge sulla base di quelle che sono le necessità definite dal dominio applicativo. Tipicamente un applicativo *web* dinamico richiede anche che sia possibile effettuare serializzazione e deserializzazione dei dati, ed è proprio a questo fine che viene introdotto il terzo ed ultimo strato, ovvero il *Repository* (termine traducibile appunto in "deposito" o "magazzino").

Nell'immagine proposta di seguito è possibile osservare una rappresentazione visiva di tale organizzazione.

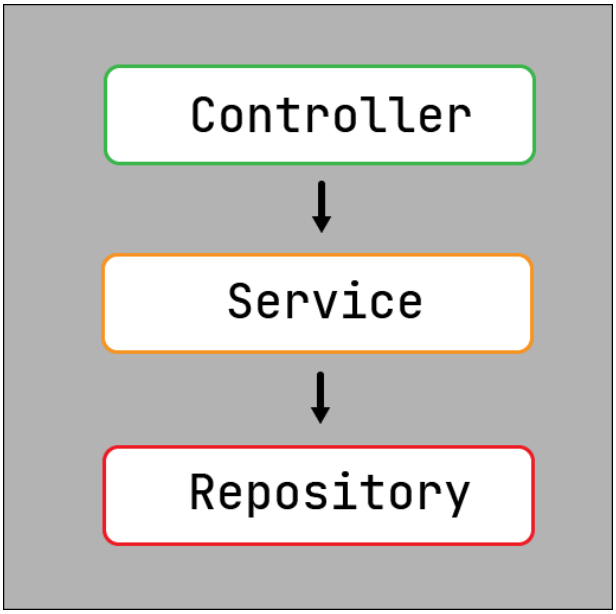


Figura 3.2: Rappresentazione visiva dell'architettura di riferimeto.

Come si può osservare le dipendenze vanno dall'alto verso il basso, quindi nel caso si considerino le classi appartenenti allo strato *Controller* queste richiamano metodi propri delle classi dello strato *Service* e, allo stesso modo, le classi appartenenti allo strato *Service* richiamano metodi delle classi dello strato *Repository*. La gestione di suddette dipendenze viene effettuata per mezzo della *dependency injection*, un *design pattern* proprio della programmazione ad oggetti utile a semplificare lo sviluppo e implementare il meccanismo dell' *inversion of control*.
Il meccanismo alla base di tale *pattern* prevede che le dipendenze di una classe vengano dichiarate esplicitamente attraverso il costruttore della stessa ma che, quando risulti necessario istanziare un oggetto concreto la risoluzione di suddette dipendenze sia delegata a un componente terzo detto *injector*.

Dal punto di vista operativo (quindi della codifica) questo si traduce in due operazioni specifiche:

- Dichiarazione esplicita delle dipendenze di una classe per mezzo del costruttore, che richiede esplicitamente un oggetto della classe cui si rivolge la dipendenza;
- Configurazione della classe (le cui modalità cambiano in base al *framework* utilizzato) per permettere alla componente *injector* di rilevare e risolvere la dipendenza dichiarata.

Tutto ciò che il programmatore deve fare è quindi dichiarare le classi che desidera coinvolgere nel meccanismo e dichiarare per mezzo dei costruttori quali sono le dipendenze proprie di una classe. La risoluzione di suddette dipendenze viene dunque delegata al *framework* che, per mezzo del meccanismo appena descritto, permette al

programmatore di concentrare le proprie risorse sull'implementazione delle logiche di dominio.
Nell'immagine che segue è possibile osservare una rappresentazione grafica del meccanismo appena descritto.

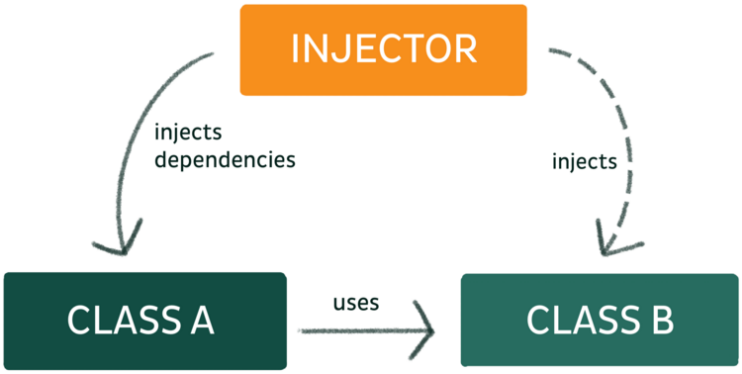


Figura 3.3: Rappresentazione visiva del meccanismo di *dependency injection*. Si osserva come vi sia una dipendenza dalla classe A alla classe B e come questa venga risolta dall'*injector* rendendo tale procedura trasparente al programmatore.
Fonte: Google immagini,*keyword*: *dependency injection visual representation.*

Tale idea, ovvero quella per cui è desiderabile ridurre al minimo gli oneri implementativi del programmatore, permea il raziocinio alla base delle tecnologie esaminate arrivando anche a rendere trasparente la manipolazione dei dati su cui viene fatta serializzazione e deserializzazione. La tecnica di programmazione che implementa tale meccanismo prende il nome di *object-relational mapping*, spesso abbreviata per mezzo della sigla ORM. Essa consiste nel modellare le classi con cui il programmatore interagisce sulla base delle entità che popolano un *database* relazionale, permettendo di astrarre i meccanismi di comunicazione con lo specifico *database* utilizzato. Volendo esemplificare, si può immaginare di voler realizzare un servizio che necessita di dare persistenza ai dati relativi ai propri utenti. Tali utenti potrebbero essere tracciati per mezzo di un codice identificativo, del proprio nome, cognome e anno di nascita. Per permettere una comunicazione trasparente con la base di dati con cui si sta lavorando è necessario unicamente modellare una classe che possegga come campi dati gli attributi dell'entità presente nel *database* (nel nostro caso quindi codice identificativo, nome, cognome e anno di nascita) e configurare (secondo i meccanismi propri del *framework* in uso) il sistema affinché venga automaticamente riconosciuta la corrispondenza tra la classe modellata e l'entità presente nel *database*. Questo approccio porta a un insieme di benefici dal punto di vista della codifica:

- La corrispondenza 1:1 tra istanza di oggetto e entità presente nel *database* non introduce cambiamenti nella rappresentazione dell'oggetto di dominio;

- Le modifiche a un oggetto concreto per mezzo del codice si riflettono automaticamente sui dati presenti nel *database*, senza che vi sia la necessità da parte del programmatore di definire procedure di aggiornamento;

- I tempi di codifica, soprattutto quelli legati all'implementazione di funzionalità utili alla comunicazione con il *database*, vengono sostanzialmente azzerati in quanto delegati al meccanismo in questione.

Come detto i concetti esposti fino ad ora sono parte integrante del funzionamento dei *framework* esaminati che differiscono per le modalità con cui questi vengono integrati. Tali differenze sono alla base dell'analisi comparativa prodotta e diventano quindi rilevanti per una comprensione efficace dei risultati ottenuti.

3.3 Implementazione in Java Spring

In questa sezione espongo l'implementazione di un servizio mediato dall'utilizzo del *framework* Java Spring. Dei 3 servizi implementati durante l'attività di *stage* viene preso come esempio il primo, ovvero quello adibito alla gestione dell'utenza. L'inizializzazione di un progetto che fa utilizzo del *framework* Java Spring viene eseguita attraverso un *tool* fornito direttamente dagli sviluppatori che permette di automatizzare la configurazione del progetto.
Suddetta configurazione permette di selezionare e modificare una serie di parametri, che vanno dal nome del progetto al *software* che si desidera usare per gestire il meccanismo della *build automation* (ad esempio Maven o Gradle). Il più importante è sicuramente quello relativo ai moduli che si desidera includere all'interno del proprio progetto. Nel caso specifico dei microservizi implementati, i moduli del *framework* più rilevanti sono:

- Spring Boot Web: modulo di base per la creazione di API REST secondo il modello a microservizi attraverso l'utilizzo del *framework*. Questo modulo implementa le funzionalità fondamentali disponibili al programmatore, tra cui il meccanismo di *dependency injection* e di *unit test*;

- Spring Data JPA: modulo utile all'implementazione del meccanismo dell'ORM. Permette di comunicare con *database* relazionali attraverso l'astrazione e l'automazione delle procedure di connessione e trasferimento dati da e verso il *database* che si decide di utilizzare. Utile solamente nel contesto dei *database* relazionali.

Questi moduli di fatto costituiscono lo scheletro fondamentale di funzionalità necessarie alla creazione di un microservizio nel contesto di Java Spring, ma ritengo sia bene sottolineare che il *framework* mette a disposizione dell'utente un ampio numero di moduli che svolgono funzioni molto differenti. Ad esempio, è possibile utilizzare il modulo Java Spring Batch per ottimizzare il *processing* di grandi moli di dati, oppure il modulo Java Spring Security per implementare funzionalità come la sanificazione dei dati in *input* per garantire che il sistema non venga violato.
Al termine dell'inizializzazione, verrà prodotto dal *tool* un *file* compresso che contiene una cartella già configurata secondo la struttura che è possibile vedere nell'immagine riportata di seguito.

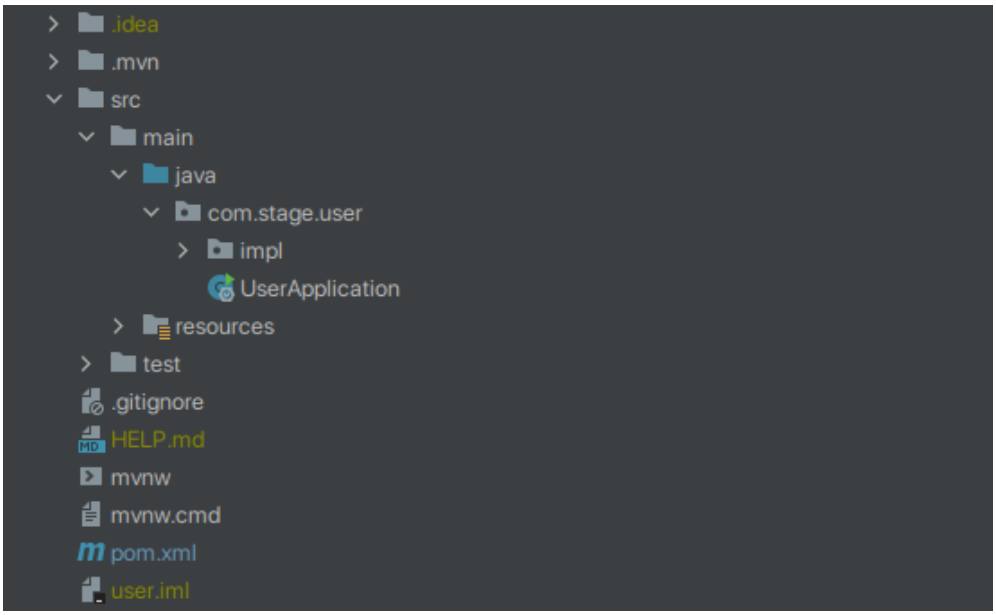


Figura 3.4: Istantanea della struttura di un progetto inizializzato con l'apposito *tool* per lo sviluppo tramite il *framework* Java Spring.

In primo luogo si può osservare che vengono generati i file di configurazione per l'integrazione con i servizi di versionamento (*.gitignore*, utile per definire i file di cui non si vuole fare storicizzazione) e *build automation* (*pom.xml*, utile a definire le dipendenze esterne del progetto). Per quanto riguarda le *directory* invece, la più importante è *src* che contiene (come suggerito anche dal nome) i file sorgenti dell'applicativo. Essa contiene (conformemente a quanto ci si aspetta nel contesto del linguaggio Java) il *package* principale, ovvero *com.stage.user*, e la cartella destinata a contenere gli *unit test* che verificano la qualità del prodotto (esaminata più avanti). Il *package* principale, a sua volta, ha due contenuti fondamentali:

- Il *file UserApplication.java*, che rappresenta l'*entry-point* per l'esecuzione dell'applicativo. Tale file viene generato automaticamente dal *tool* al momento della creazione del progetto;
- Il *subpackage impl* (abbreviazione del termine inglese *implementation*) da me definito. Esso contiene l'implementazione delle classi che modellano il servizio.

È inoltre possibile osservare che è presente una cartella denominata *resources* che contiene *file* di configurazione aggiuntivi per l'applicativo sviluppato. Il cuore pulsante dell'ambiente di lavoro per il microservizio in questione è la cartella *impl*, che si struttura come si può osservare nella figura riportata di seguito.

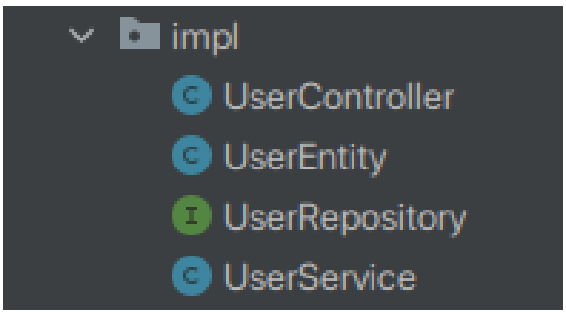


Figura 3.5: Istantanea della struttura relativa alla cartella *impl*, che contiene le implementazioni dei diversi strati della *Layered architecture* corrispondente.

Ciascuno dei file implementa uno degli strati previsti dall'architettura presentata nella sezione 4 e la cui nomenclatura rimanda direttamente alle funzionalità che questi implementano con la sola eccezione fatta per il *file UserEntity*. Questo contiene l'implementazione di una classe (*UserEntity* appunto) utile per abilitare il meccanismo dell'ORM. Un oggetto della classe *UserEntity* possiede come campi dati gli attributi che si possono trovare anche nel *database* su cui si effettua persistenza dei dati, coadiuvato dai metodi per la lettura e la modifica dei valori corrispondenti. Il *file* si struttura nella maniera seguente:

```
1   @Entity
2   @Table
3   public class UserEntity {
4
5       @Id
6       @SequenceGenerator(
7               name = "user_sequence",
8               sequenceName = "user_sequence",
9               allocationSize = 1
10      )
11      @GeneratedValue(
12              strategy = GenerationType.SEQUENCE,
13              generator = "user_sequence"
14      )
15      private Long id;
16      private String name;
17      private String email;
18
19      public UserEntity() {}
20
21      public UserEntity(Long id, String name, String email) {
    ... }
22
23      public UserEntity(String name, String email) { ... }
24
25      ( ... Getters e setters contestuali ... )
26  }
```

Codice sorgente 3.1: Contenuto del file UserEntity.java

La prima cosa da osservare è che il *framework* implementa molte delle proprie funzionalità per mezzo della sintassi nota come "annotazione" (in originale, *Java Annotations*). Una annotazione, nel contesto della programmazione ad oggetti in Java, è un espressione che permette di implementare funzionalità specifiche per classi, metodi e attributi per mezzo della sintassi *@NomeAnnotazione* dove "NomeAnnotazione" viene sostituito con il nome effettivo della funzionalità che si desidera implementare. Calando quanto appena detto nel contesto del codice della classe *UserEntity* presentato, si può osservare che questo meccanismo permea l'intera struttura del *file* e ciò rimane vero anche per le classi che vengono viste in seguito. I primi due decoratori, *@Entity* e *@Table*, servono per rendere esplicita la corrispondenza 1:1 tra le entità di cui viene fatta persistenza nel *database* e gli oggetti che la classe modella e attivare le funzionalità che rendono trasparenti le operazioni da e verso la base di dati stessa. All'interno della classe si possono osservare poi altri tre decoratori applicati al campo dati *id*. Questi sono nell'ordine:

- *@Id*: questa annotazione permette di stabilire una corrispondenza univoca tra il campo dati dell'oggetto e l'identificatore univoco dell'entità nel *database*;
- *@SequenceGenerator e @GeneratedValue*: queste annotazioni servono per rendere automatica la generazione e la gestione per gli identificatori univoci di un entità presente nel *database* e mappata dall'oggetto.

La classe appena presentata è necessaria qualora si desideri gestire la comunicazione con una base di dati per mezzo del meccanismo dell'ORM, ma questo non è necessario per il corretto funzionamento dell'applicativo. I *file* restanti, come detto, modellano l'architettura descritta nella sezione precedente. Seguendo quindi il flusso delle dipendenze (dall'alto verso il basso, si veda la figura 3.6) il primo componente esaminato è quello del file *UserController*. Questo (tralasciando dichiarazioni di *import* contestuali) si struttura nella maniera seguente:

```
1   @RestController
2   @RequestMapping(path="user")
3   public class UserController {
4       private final UserService userService;
5
6       @Autowired
7       public UserController(UserService userService) { ... }
8
9       @GetMapping
10      public List<UserEntity> getUsers() { ... }
11
12      @PostMapping
13      public Long registerNewUser(@RequestBody UserEntity user)
    { ... }
14
15      @DeleteMapping(path = "{userId}")
16      public void deleteUser(@PathVariable("userId") Long id) {
    ... }
17
18      @PutMapping
19      public void updateUser(@RequestBody UserEntity user) {
... }
20  }
```

Codice sorgente 3.2: Contenuto del file UserController.java

In primo luogo, l'annotazione *@RestController* configura per mezzo del *framework* la classe *UserController* per far sì che questa riceva le chiamata HTTP in ingresso e siano le procedure in essa definite a gestire le operazioni da svolgere. L'annotazione *@RequestMapping(path="user")* serve invece per identificare il *path* a cui questo specifico *Controller* fa riferimento. Internamente la classe ha un unico campo dati, ovvero *userService* che corrisponde anche all'unica dipendenza che la classe presenta. Come accennato nella sezione precedente la gestione delle dipendenze viene delegata al meccanismo della *dependency injection*. Osservando il costruttore della classe si può notare subito come tale meccanismo viene implementato nello specifico dal *framework*: il metodo che funge da costruttore viene decorato con l'annotazione *@Autowired*, per indicare al *framework* che la risoluzione delle dipendenze deve essere automatizzata, e nella dichiarazione dei parametri viene esplicitamente richiesto un oggetto della classe appropriata. Questi due passaggi sono sufficienti per permettere al *framework* di fornire un'istanza concreta del tipo richiesto, di fatto nascondendo il funzionamento del meccanismo al programmatore.
I metodi restanti (*getUsers, registerNewUser, deleteUser e updateUser*) mappano quattro differenti chiamate HTTP che possono essere ricevute. In particolare:

- *getUsers* mappa le chiamate HTTP GET tramite l'annotazione *@GetMapping*;
- *registerNewUser* mappa le chiamate HTTP POST tramite l'annotazione *@PostMapping*;
- *deleteUser* mappa le chiamate HTTP DELETE tramite l'annotazione *@DeleteMapping*;
- *updateUser* mappa le chiamate HTTP PUT tramite l'annotazione *@PutMapping*.

Nel contesto di un'applicazione che implementa REST API, può accadere che una parte dei dati sia concatenata direttamente all'interno del *path* oppure che questi vengano inseriti come parte del corpo (*body*) della richiesta HTTP. Nel caso in cui si desideri recuperare tali dati è possibile farlo tramite apposite annotazioni: se il valore è posto in coda al *path* questo può essere recuperato per mezzo dell'annotazione *@PathVariable* indicando tra parentesi il nome del valore associato, mentre se il valore è contenuto direttamente nel corpo della richiesta è possibile ottenerlo per mezzo dell'annotazione *@RequestBody* che provvede a mapparlo a un oggetto di dominio.

La classe *UserController* appena esaminata si limita nella sua implementazione a richiamare metodi della classe membro *UserService* che è cosi strutturata:

```
1   @Service
2   public class UserService {
3       private final UserRepository userRepository;
4
5       @Autowired
6       public UserService(UserRepository userRepository) { ... }
7       public List<UserEntity> getUsers() { ... }
8       public Long addNewUser(UserEntity user) { ... }
9       public void deleteUser(Long id) { ... }
10
11      @Transactional
12      public void updateUser(UserEntity modifiedUser) { ... }
13  }
```

Codice sorgente 3.3: Contenuto del file UserService.java

Si può immediatamente osservare come la struttura del codice in questo caso sia molto simile a quella della classe *UserController*. Le principali differenze sono due:

- L'annotazione presente sopra la dichiarazione di classe è cambiata contestualmente al ruolo ricoperto dalla classe stessa. Se infatti la classe *UserController* utilizzava l'annotazione *Controller*, in questo caso l'annotazione richiesta è quella *Service*. Coerentemente a quanto già detto nella sezione precedente, tale annotazione serve a identificare nel contesto del *framework* una classe che effettua operazioni di calcolo sui dati ricevuti;

- L'annotazione *@Transactional* viene utilizzata per decorare il metodo *updateUser* e abilitando le funzionlità che permettono di modificare i dati contenuti nella base di dati su cui si effettua persistenza dei dati semplicemente operando modifiche su oggetti istanziati. Normalmente, sarebbe necessario gestire manualmente le operazioni di lettura e scrittura da e verso il *database*, ma per mezzo di questa annotazione il programmatore viene sollevato dall'onere di doverlo fare, semplificando la codifica.

L'unica dipendenza che questa classe ha è nei confronti della classe *UserRepository*, la cui implementazione viene riportata di seguito:

```
1    @Repository
2    public interface UserRepository extends JpaRepository<
     UserEntity, Long> {
3
4        @Query( ... )
5        Optional<UserEntity> findUserByEmail(String email);
6    }
```

Codice sorgente 3.4: Contenuto del file UserRepository.java

In questo caso si può osservare che vi è un'inesattezza nelle mie affermazioni precedenti. Quella riportata sopra infatti non è una classe, ma un interfaccia (che nel contesto del linguaggio Java rappresenta un contratto che definisce dei comportamenti che una classe deve implementare). Ho introdotto questa inesattezza proprio per portare alla luce la prima e più importante caratteristica di questo aspetto del *framework*: non è neccessario definire una classe completa, ma estendendo un'interfaccia fornita dal *framework* (*JpaRepository<UserEntity, Long>*) le classi necessarie vengono automaticamente implementate. Osservando le annotazioni presenti in questo caso, ci si può accorgere ancora una volta che la classe viene decorata con l'annotazione *@Repository* (il raziocinio alla base di ciò è il medesimo visto per le due classi precedenti), e che internamente la classe implementa un unico metodo. È proprio su questo aspetto che vorrei soffermare la mia attenzione: il metodo riportato ha come scopo quello di implementare una *query* specifica (ed è a tal fine decorato con l'annotazione *@Query* seguita tra parentesi dalla formulazione in SQL della medesima), ma è assolutamente opzionale e non necessario per il corretto funzionamento della classe prodotta. Nello specifico, il *framework* si occupa di implementare tutti i metodi per le operazioni di lettura e scrittura effettuabili sul *database*, ed è avanzato a sufficienza da permettere anche di definire procedure senza la definizione contestuale della *query* specifica. Un esempio in questo senso potrebbe essere rappresentato dalla volontà di richiedere che vengano ritornati tutti gli identificativi univoci degli utenti che hanno un nome specifico. È possibile definire tale procedura semplicemente dichiarando un metodo nella maniera seguente *List<UserEntity> getIdByUserName(String name)*. Con "semplicemente"

intendo sottolineare che non è necessario né aggiungere alcun decoratore né definire il metodo. La dichiarazione dello stesso è sufficiente a renderlo disponibile all'utente della classe.

Quanto detto finora riguarda l'implementazione del servizio *User*, quindi la scrittura del codice che effettivamente implementa le funzionalità richieste.
Per verificarne l'effettivo funzionamento, ho implementato contestualmente anche i *test* di unità associati per mezzo della *utility* JUnit. Di seguito riporto il codice sorgente relativo unicamente ai *test* effettuati sulla classe *UserService*, in quanto tutte le classi che implementano i *test* utilizzano le medesime funzionalità e struttura.

```
1     class UserControllerTests {
2
3     private UserController userController;
4     private UserService userServiceMock;
5
6     @BeforeEach
7     void setUp() { ... }
8
9     @Test
10    void shouldGetUsersTest() { ... }
11
12    ( ... implementazione degli altri test per i metodi della
      classe UserController ...)
13 }
```

Codice sorgente 3.5: Contenuto del file UserControllerTests.java

Come per le componenti viste in precedenza, anche in questo caso le funzionalità principali vengono implementate utilizzando il meccanismo delle annotazioni. Si può osservare che internamente la classe *UserControllerTests* presenta due campi dati: il primo di questi rappresenta la classe di cui si vuole effettuare la verifica (in questo caso *UserController*) mentre il secondo rappresenta invece un *mock* della classe *UserService* che verrà utilizzato per istanziare la classe *UserController* che la richiede come dipendenza. Il motivo per cui si utilizza un *mock*(ovvero un oggetto che ha il medesimo tipo della classe *UserService* ma il cui comportamento viene definito *ad hoc* per renderne prevedibile il comportamento) invece di un istanza concreta del tipo *UserService* è quello di permettere di svolgere i *test* sulla classe specifica isolandola e rendendo indipendente il suo comportamento da quello delle classi nei confronti della quale questa presenta delle dipendenze. Ciò permette di identificare e risolvere i problemi di natura implementativa legati alla sola classe in esame, riducendo quindi la complessità generale dei *test* e aumentandone il grado di efficacia. Successivamente viene definito un metodo che, per mezzo dell'annotazione *@BeforeEach*, viene eseguito prima che ogni *test* venga svolto in modo da configurare le classi in maniera opportuna. Infine, per mezzo dell'annotazione *@Test*, vengono implementati i metodi relativi ai casi di *test* specifici, che verificano la qualità del codice sorgente prodotto. Sono riuscito a verificare la bontà delle funzionalità prodotte attraverso la corretta implementazione di tutti i *test* previsti, dimostrando quindi che le funzionalità richieste implementano correttamente le logiche previste.
Onde evitare incomprensioni, specifico che i dati riportati si riferiscono unicamente ai *test* di unità (che verificano il corretto funzionamento di una logica esaminata però atomicamente), mentre non sono stati richiesti e non sono previsti *test* di integrazione

tra le componenti. I risultati ottenuti a valle dell'implementazione dei *test* sono i seguenti:

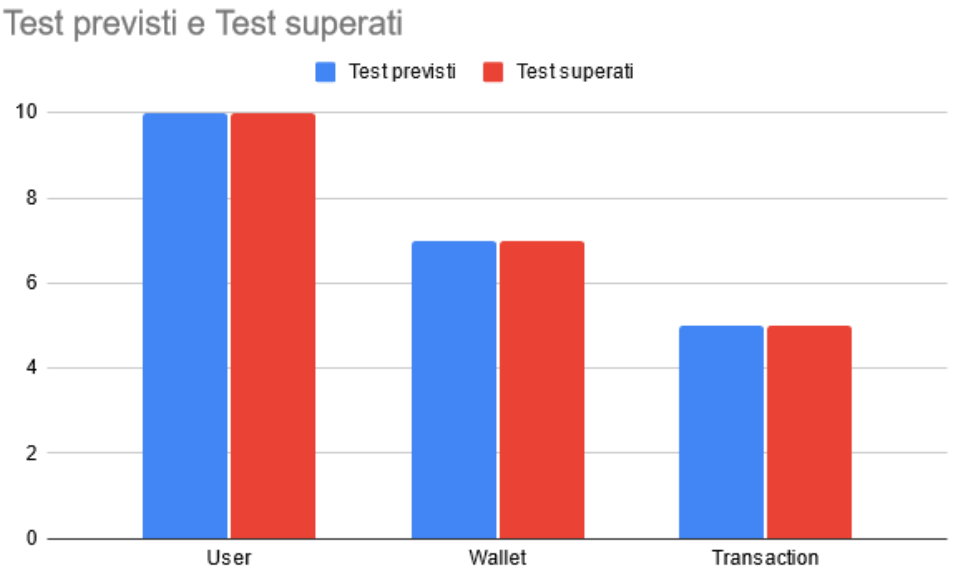


Figura 3.6: Istogramma relativo ai *test* implementati e superati nel contesto Spring.

3.4 Implementazione in NestJS

L'inizializzazione di un nuovo progetto sviluppato per mezzo del *framework* NestJS è mediata da un *tool* da riga di comando fornito direttamente dagli sviluppatori del *framework*. Nelle intenzioni degli sviluppatori, ogni operazione relativa all'aggiunta di nuove componenti o modifica del progetto deve essere moderato dall'utilizzo di questo *tool*, di cui viene riportato l'*help* di seguito:

```
$ nest --help
Usage: nest <command> [options]

Options:
  -v, --version                                   Output the current version.
  -h, --help                                      Output usage information.

Commands:
  new|n [options] [name]                          Generate Nest application.
  build [options] [app]                           Build Nest application.
  start [options] [app]                           Run Nest application.
  info|i                                          Display Nest project details.
  add [options] <library>                         Adds support for an external library to your project.
  generate|g [options] <schematic> [name] [path]  Generate a Nest element.
    Schematics available on @nestjs/schematics collection:

    name            alias        description
    application     application  Generate a new application workspace
    class           cl           Generate a new class
    configuration   config       Generate a CLI configuration file
    controller      co           Generate a controller declaration
    decorator       d            Generate a custom decorator
    filter          f            Generate a filter declaration
    gateway         ga           Generate a gateway declaration
    guard           gu           Generate a guard declaration
    interceptor     itc          Generate an interceptor declaration
    interface       itf          Generate an interface
    middleware      mi           Generate a middleware declaration
    module          mo           Generate a module declaration
    pipe            pi           Generate a pipe declaration
    provider        pr           Generate a provider declaration
    resolver        r            Generate a GraphQL resolver declaration
    service         s            Generate a service declaration
    library         lib          Generate a new library within a monorepo
    sub-app         app          Generate a new application within a monorepo
    resource        res          Generate a new CRUD resource
```

Figura 3.7: Un illustrazione dell'*help* relativo al *tool* distribuito insieme al *framework* NestJS

L'architettura proposta in questo senso è la medesima esaminata ed esposta nella sezione precedente, pertanto ciascun servizio viene strutturato secondo il medesimo raziocinio e implementato con le stesse modalità. Dopo aver configurato il progetto è possibile osservare la presenza dei *file* di configurazione *package.json* e *tsconfig.json* (che sono i più importanti e quindi vengono approfonditi), utili a configurare l'ambiente di lavoro per la corretta esecuzione dell'applicativo tramite il *runtime* Node.js. Tali file, in un contesto estraneo all'utilizzo del *framework*, andrebbero configurati manualmente (introducendo quindi possibili errori e comportamenti indesiderati) mentre attraverso l'utilizzo di NestJS la configurazione viene automatizzata e resa trasparente al programmatore. In particolare, il primo tra questi è necessario proprio per definire i parametri legati al progetto nella sua interità e permettere al *runtime* di risolvere le dipendenze che il progetto presenta nei confronti di librerie esterne, mentre il secondo è necessario per permettere la corretta interpretazione del codice Typescript implementato. La struttura della *directory* è riportata di seguito:

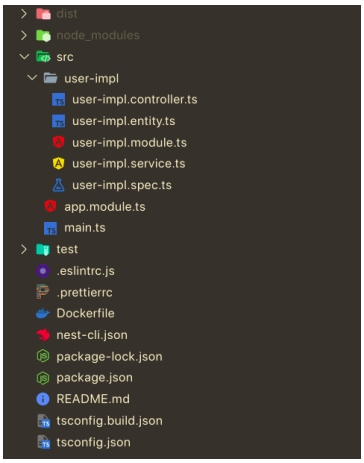


Figura 3.8: Una rappresentazione visiva della struttura relativa alla *directory* prodotta dal *tool* di configurazione dopo che viene eseguito il comando *nest new application*.

Similmente a quanto detto nella sezione precedente per il *framework* Spring la *directory* più importante è *src*, che contiene i sorgenti dei *file* che effettivamente implementano le funzionalità desiderate. Dal punto di vista del contenuto, si possono osservare due *file* (*app.module.ts* e *main.ts*) e una sottocartella (*user-impl*).
La prima e più importante differenza rispetto a Java Spring risiede nella presenza del *file app.module.ts*. Questo, nonostante sia un *file* con estensione *.ts*, è in realtà utilizzato per implementare parte della configurazione relativa al *framework*. Sebbene questa caratteristica sia apparentemente insignificante, è in realtà uno dei punti fondamentali che differenziano le due tecnologie: il *framework* implementa parte della configurazione per mezzo di codice sorgente che, dal punto di vista strutturale, non viene chiaramente separato dal codice che invece implementa logiche di dominio.
La cartella appena descritta si struttura nella maniera seguente:

Figura 3.9: Una rappresentazione visiva della struttura relativa alla *directory src* prodotta dal *tool* di configurazione.

Lo stesso problema si presenta anche all'interno della sottocartella, all'interno della quale è possibile osservare che oltre ai *file* che implementano le logiche di dominio e i *layer* dell'architettura è possibile osservare che ci sono altri *file* (come ad esempio *user-impl-module.ts*). Questo ricopre una funzione analoga al primo, e quindi è necessario per configurare il *framework* per garantirne il corretto funzionamento. Il raziocino che gli sviluppatori hanno seguito nel prendere questa decisione è quello di presentare una struttura simile a quella di altri *framework* disponibili per lo sviluppo di interfacce utente in ambiente Node.js. Riproponendo la medesima struttura, l'obiettivo degli sviluppatori è quello di permettere la creazione di un ambiente di sviluppo per *back-end* e *front-end* di un applicativo *web* in cui i linguaggi e i meccanismi seguiti dai *framework* utilizzati sono i medesimi, alleggerendo quindi il programmatore dall'onere di lavorare con tecnologie e strutture di progetto differenti. In altre parole, unificando quanto più possibile gli strumenti che il programmatore utilizza durante lo sviluppo diventa (idealmente) meno probabile che questo commetta errori di natura logica. Le conseguenze di tale approccio sono, come detto, un carico maggiore in termini di configurazione per il programmatore stesso (che invece, per i motivi prima esposti, è indesiderabile).

L'implementazione delle logiche di dominio è effettuata interamente all'interno della sottocartella *user-impl*, la cui struttura è visibile nella figura precedente. Come per il *framework* Spring, la mia disamina dei *file* contenuti in questa cartella comincia da *user-impl.entity.ts* che ha il medesimo scopo del *file UserEntity* nel contesto di Spring, ovvero modellare le entità che popolano il *database* modellandole come oggetti per abilitare le funzionalità dell'ORM. Di seguito ne viene riportato il codice:

```
1   @Entity()
2   export class UserEntity {
3       @PrimaryGeneratedColumn()
4       id: number;
5
6       @Column()
7       name: string;
8
9       @Column()
10      email: string;
11  }
```

Codice sorgente 3.6: Contenuto del file user-impl.entity.ts

La prima osservazione da fare è che anche questo *framework* implementa molti dei propri meccanismi per mezzo della medesima sintassi dell'annotazione (con la sola eccezione che, per permettere l'integrazione della funzionalità per mezzo del linguaggio Typescript è sempre necessario aggiungere le parentesi utili a contenere parametri anche in assenza di questi per mezzo della notazione "()"). La prima di queste che si può osservare è *@Entity*, che è utile per segnalare al *framework* la volontà di utilizzare la classe definita per modellare entità di dominio contenute nel *database*.
Le altre annotazioni sono *@PrimaryGeneratedColumn* e *@Column*, utili per identificare rispettivamente il campo dati che funge da chiave primaria e i campi dati che mappano gli attributi della singola entità. Interessante osservare l'assenza di costruttore e metodi per la modifica dei campi dati di un istanza dell'oggetto, caratteristica legata ai meccanismi di funzionamento del linguaggio Typescript che si differenzia in questo da Java. La seconda osservazione che è importante fare risiede nella presenza del file *user-impl.module.ts* di cui viene riportato il codice di seguito:

```
@Module({
    imports: [TypeOrmModule.forFeature([UserEntity])],
    controllers: [UserImplController],
    providers: [UserImplService]
})
export class UserImplModule {}
```

Codice sorgente 3.7: Contenuto del file user-impl.module.ts

Il codice appena riportato, anche se all'apparenza breve, è denso di contenuto informativo, ed è necessario porre molta attenzione a ciò che riporta. L'annotazione *@Module* serve per dichiarare la volontà di istanziare una classe (nel caso da me riportato *UserImplModule*) che funge appunto da modulo. Nel contesto di NestJS un modulo è un'entità che permette di abilitare il corretto funzionamento dell'architettura proposta (una *Layered architecture* a 3 strati, *Controller, Service e Repository*). L'annotazione *@Module* serve quindi per dichiarare, attraverso la definizione di un oggetto JSON (Javascript Object Notation) apposito, i seguenti parametri:

- Quali funzionalità importare da libreire esterne, per mezzo del campo *imports*;
- Le classi che implementano le funzionalità proprie della componente *Controller*, per mezzo del campo *controllers*;
- Le classi che implementano le funzionalità proprie della componente *Service*, per mezzo del campo *providers*.

La corretta definizione di questi parametri è necessaria per abilitare i meccanismi di ORM e *dependency injection* che il *framework* utilizza, e rappresenta una differenza sostanziale tra NestJS e Java Spring, poiché nel secondo tali meccanismi non devono essere configurati dal programmatore ma sono automaticamente abilitati dal *framework* quando venivano rilevate le annotazioni contestuali. In questo caso invece, al fine di permettere una configurazione "a grana fine" dell'ambiente, la decisione è quella di delegare al programmatore stesso l'onere di decidere quali classi e funzionalità debbano essere abilitate a svolgere ruoli specifici.

Terminata la configurazione, procedo quindi ad esaminare il contenuto dei *file* che implementano propriamente gli strati dell'architettura partendo da *user-impl.controller.ts* che implementa il *Controller layer* e il cui codice sorgente viene riportato di seguito:

```
1   @Controller('user')
2   export class UserImplController {
3
4       constructor(private userImplServce: UserImplService) {};
5
6       @Get()
7       async getUsers(): Promise<UserEntity[]> { ... }
8
9       @Post()
10      async registerNewUser(@Body() user: UserEntity): Promise<
    number> { ... }
11
12      @Delete(':id')
13      async deleteUser(@Param('id') id: number) { ... }
14
15      @Put()
16      async updateUser(@Body() user: UserEntity) { ... }
17
18  }
```

Codice sorgente 3.8: Contenuto del file user-impl.controller.ts

La prima annotazione, ovvero *@Controller('user')*, similmente a quanto accade nel contesto di Java Spring serve per fare in modo che il *framework* indirizzi le chiamate HTTP in ingresso alla classe decorata (il parametro *'user'* svolge la medesima funzione del parametro con il medesimo ruolo passato all'annotazione *@RequestMapping* nel contesto di Spring). Successivamente, è possibile osservare che il costruttore non viene decorato con nessuna annotazione per abilitare il meccanismo di *dependency injection* in quanto questo è già stata abilitato per mezzo del *file user-impl.module.ts* visto in precedenza. Infine, è possibile osservare che i metodi vengono decorati anche in questo caso per mezzo di annotazioni che mappano le possibili chiamate HTTP che la classe può trovarsi a dover gestire. In particolare:

- *getUsers* mappa le chiamate HTTP GET tramite l'annotazione *@Get*;
- *registerNewUser* mappa le chiamate HTTP POST tramite l'annotazione *@Post*;
- *deleteUser* mappa le chiamate HTTP DELETE tramite l'annotazione *@Delete*;
- *updateUser* mappa le chiamate HTTP PUT tramite l'annotazione *@Put*.

Le annotazioni *@Body* e *@Param* svolgono le medesima funzione delle corrispondenti *@RequestBody* e *@PathVariable* nel contesto Spring (quindi permettere di recuperare dati contenuti rispettivamente nel corpo della richiesta e in coda al *path*).

Anche in questo caso la classe *UserImplController* utilizza le funzionalità implementate dai metodi della classe che implementa le logiche di dominio, ovvero *UserImplService*, e il cui codice viene riportato di seguito:

```
1   @Injectable()
2   export class UserImplService {
3
4       constructor(@InjectRepository(UserEntity) private
    userRepository: Repository<UserEntity>) {};
5
6       async getAllUsers(): Promise<UserEntity[]> { ... }
7
8       async registerNewUser(user: UserEntity): Promise<number>
    { ... }
9
10      async deleteUser(id: number) { ... }
11
12      async updateUser(user: UserEntity) { ... }
13
14  }
```

Codice sorgente 3.9: Contenuto del file user-impl.service.ts

L'annotazione *@Injectable* è utile per segnalare al *framework* la volontà di utilizzare questa classe dovunque sia richiesto per mezzo del meccanismo della *dependency injection* utilizzato. Internamente, la classe presenta lo stesso comportamente della controparte sviluppata per mezzo di Java Spring, pertanto i metodi effettuano modifica, serializzazione e deserializzazione dei dati per mezzo della classe *UserRepository*, che viene fornita come dipendenza tramite un meccanismo che invece è differente. Esso viene implementato per mezzo dell'annotazione *@InjectRepository(UserEntity)*, che richiede al componente preposto (quello adibito all'abilitazione del meccanismo dell'ORM) di istanziare e fornire una classe che permette di comunicare con il *database* (e che quindi rappresenta, dal punto di vista architetturale, l'implementazione dello strato *Repository*).
Tale classe, a differenza di Java Spring, non richiede di essere nemmeno dichiarata in questo contesto ma tramite la configurazione che avviene per mezzo del *file user-impl.module.ts* questa viene modellata e istanziata in maniera totalmente trasparente al programmatore.
Ritengo sia molto importante sottolineare (per meglio contestualizzare i risultati esposti nel prossimo capitolo) che il modulo che implementa il meccanismo dell'ORM non è sviluppato nativamente dallo stesso *team* che propone il *framework* NestJS, ma è invece integrato a partire da una libreria esterna. Tale integrazione non è forzata ed è possibile decidere liberamente di utilizzare una libreria che implementa il medesimo meccanismo in maniera differente. Tutto ciò che è richiesto fare per modificare tale dipendenza del progetto è, dopo aver provveduto a recuperare e integrare il codice sorgente necessario, configurare diversamente il campo *imports* contenuto nel *file user-impl.module.ts*. Nell'esempio da me riportato (e nell'applicativo da me prodotto) la libreria che implementa tale funzionalità si chiama TypeORM e la scelta di utilizzare tale libreria si riflette sulla configurazione del campo dati precedentemente citato con la direttiva *TypeOrmModule.forFeature([UserEntity])* (interpretabile come "desidero abilitare le funzionalità di ORM proposte dalla libreria TypeORM per la serializzazione e la deserializzazione di dati che vengono modellati secondo la struttura degli oggetti della classe *UserEntity*").

L'ultimo *file* contenuto nella sottocartella *user-impl* che è necessario che io tratti è *user-impl.spec.ts* il quale implementa i *test* di unità relativi al servizio, e il cui codice viene riportato di seguito:

```
    describe('UserImplService', () => {

        let userService: UserImplService;
        let userController: UserImplController;

        beforeEach(async () => { ... });

        it('GET should return all users', async () => { ... });

});
```

Codice sorgente 3.10: Contenuto del file user-impl.spec.ts

A differenza di quanto visto nel contesto di Java Spring i *test* non vengono implementati per mezzo di una classe. I meccanismi che regolano il funzionamento delle procedure di *test* invece sono i medesimi: vengono pertanto dichiarati come variabili la classe che si desidera testare e un *mock* della dipendenza che questa presenta, insieme a un metodo utile per l'opportuna configurazione preliminare di queste da svolgere prima di ogni *test* seguita dalla dichiarazione effettiva dei metodi che implementano propriamente i casi di *test*.

I risultati ottenuti a valle dell'implementazione dei *test* sono i seguenti:

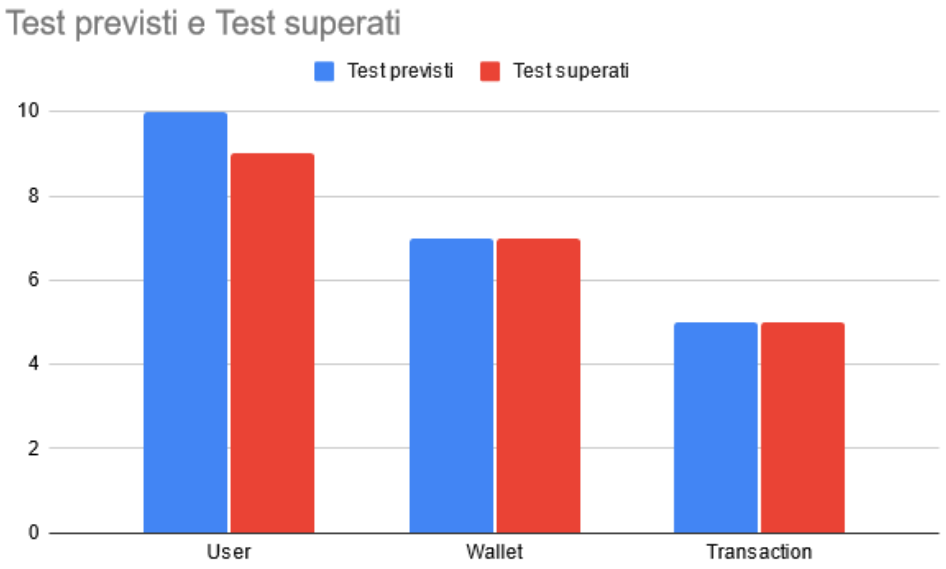


Figura 3.10: Istogramma relativo ai *test* implementati e superati nel contesto NestJS.

Sono riuscito a verificare la bontà delle funzionalità prodotte eccezion fatta per un singolo metodo. Il proponente si è comunque detto soddisfatto dei risultati ottenuti e non considera questa mancanza tale da invalidare la qualità dell'intero lavoro svolto.

3.5 Applicativi prodotti: una visione ad alto livello

In termini di risultato finale, quanto prodotto può essere rappresentato nella maniera seguente:

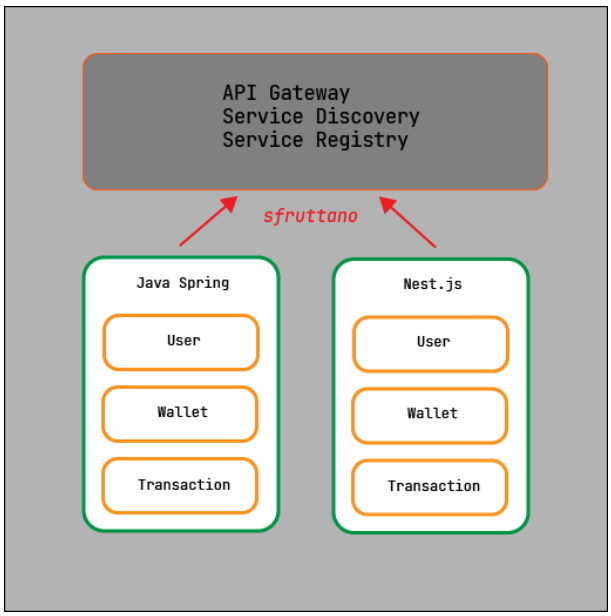


Figura 3.11: Immagine rappresentativa dei prodotti finali. Il riquadro superiore (colorato in grigio) rappresenta le componenti utilizzate per abilitare la comunicazione tra i microservizi ma non implementate. La loro trattazione è disponibile nella sezione in appendice.

Il lavoro sinergico dei servizi prodotti implementa le funzionalità previste dai caso d'uso, e i risultati prodotti sono i medesimi indipendentemente da quale versione (in relazione alla tecnologia usata) si decide di utilizzare. Questo si traduce nella possibilità di indirizzare le chiamate HTTP di interesse verso uno dei servizi a scelta, ottenendo una risposta coerente indipendentemente dal servizio cui si rivolgono suddette chiamate. Questo fatto, ovvero la totale equivalenza in termini funzionali dei servizi, permette di osservare come le tecnologie utilizzate concorrano a risolvere le medesime tipologie di problemi e l'analisi comparativa prodotta è utile proprio per capire in quali casistiche una è preferibile all'altra.
In termini di materiale propriamente prodotto di seguito vengono riportati i dati relativi al numero di *file* sorgenti e di configurazione prodotti e relative righe di codice contenute in essi, corredati da un infografica che permette di valutare "a colpo d'occhio" le differenze presenti.

Tabella 3.2: Dati relativi all'implementazione del servizio tramite il *framework* Spring

Metrica di riferimento	**Valore**
File sorgenti prodotti	24
File di configurazione prodotti	9
Righe di codice (*test* inclusi)	932

Tabella 3.3: Dati relativi all'implementazione del servizio tramite il *framework* NestJS

Metrica di riferimento	**Valore**
File sorgenti prodotti	21
File di configurazione prodotti	16
Righe di codice (*test* inclusi)	618

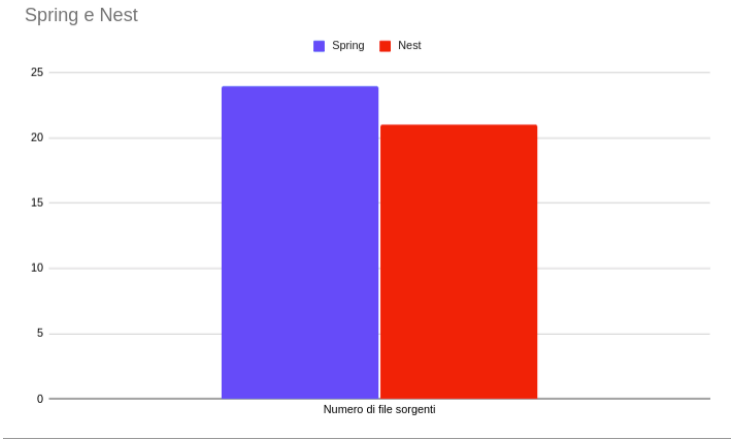


Figura 3.12: Un istogramma relativo al numero di *file* sorgenti prodotti.

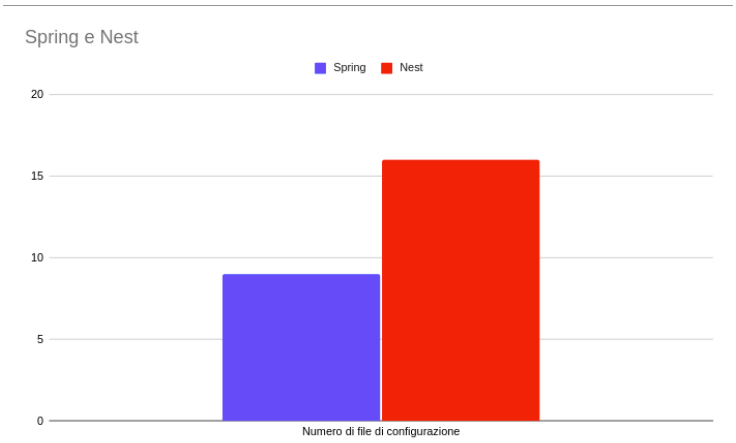


Figura 3.13: Un istogramma relativo al numero di *file* di configurazione prodotti.

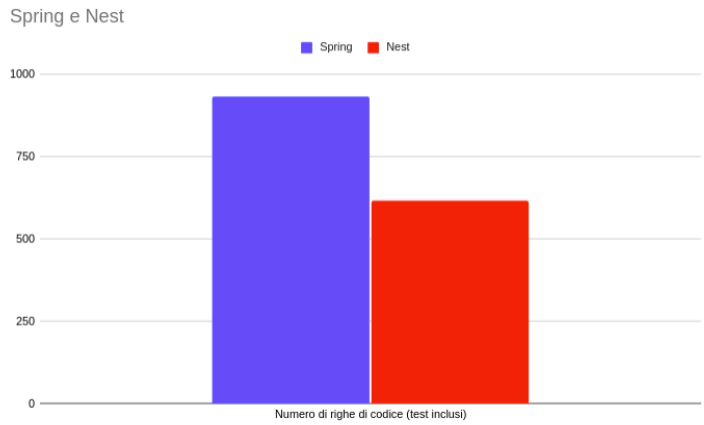


Figura 3.14: Un istogramma relativo al numeri di righe di codice prodotte.

A partire dai dati riportati le considerazioni che possono essere fatte riguardano principalmente le differenze tra gli oneri di configurazione e di implementazione che i due *framework* impongono all'utente. Se da un lato infatti Spring richiede che molti meccanismi vengano gestiti direttamente tramite il codice, dall'altro rende lo sviluppo più solido gestendo i dati "a grana fine" tramite il meccanismo delle annotazioni visto nelle sezioni precedenti. Differentemente, NestJS richiede invece che vi sia una configurazione dei meccanismi che regolano il funzionamento del codice maggiore (si veda la figura di confronto tra il numero di file di configurazione), permettendo quindi al programmatore di implementare meno codice relativo alle logiche di dominio e gestire in maniera separata il modo in cui questo viene effettivamente reso funzionale ed efficace.
Ritengo sia importante sottolineare che i dati riportati non evidenziano una caratteristica significativa: nel contesto di NestJS la distinzione tra i *file* di configurazione e quelli che implementano effettivamente il codice relativo alle logiche di dominio non presentano una divisione chiara, ma parte della configurazione avviene proprio all'interno dei *file* sorgenti. Sebbene questo in apparenza possa sembrare un dato irrilevante, costituisce in realtà un punto dolente nell'utilizzo del *framework*, non permettendo di rendere immediatamente chiara la funzione che alcuni *file* ricoprono.
Ho esaminato le informazioni appena riportate, in una forma congrua e approfondita, nell'analisi comparativa prodotta utile a confrontare (anche in relazione agli aspetti teorici visti durante la prima parte dell'attività) i due *framework* e capire quale dei due sia più adatto al contesto di sviluppo eventualmente esaminato.

Capitolo 4

Conclusioni

4.1 Obiettivi raggiunti

Concluse le attività di sviluppo dei microservizi attesi mi sono rivolto al mio *tutor* aziendale per una valutazione sul lavoro svolto, per discutere l'analisi comparativa prodotta e valutare gli obiettivi raggiunti. Riporto di seguito in forma tabellare gli obiettivi fissati dal proponente nel piano di lavoro corredati da un giudizio circa il raggiungimento (o meno) dell'obiettivo specifico prima di addentrarmi nella valutazione vera e propria fornita durante suddetto colloquio.

Sebbene il proponente si sia ritenuto soddisfatto del mio operato, non sono riuscito a verificare attraverso un *test* di unità il corretto funzionamento di uno dei metodi in ambiente NestJS, di fatto non permettendomi di dire che i microservizi sono stati implementati con una percentuale di superamento pari a 100.
L'unico obiettivo personale misurabile che mi sono posto a livello personale è di completare l'implementazione dei microservizi con una percentuale di superamento pari a 90, di fatto raggiunto. Gli altri obiettivi, non misurabili, sono da me considerati come raggiunti grazie al riscontro ricevuto da parte del proponente, che durante il nostro ultimo colloquio ha verificato e approvato la mia maturazione rispetto alle competenze acquisite senza sollevare dubbi circa la bontà del lavoro di ricerca e formazione sulle tematiche proposte da me affrontato.

Tabella 4.1: Tabella riassuntiva dei risultati raggiunti.

Tipologia	**Obiettivo**	**Stato**
Obbligatorio	Acquisizione di competenze sulle tematiche affrontate.	✓
Obbligatorio	Capacità di raggiungere gli obiettivi richiesti in autonomia seguendo il cronoprogramma.	✓
Obbligatorio	Completare l'implementazione dei microservizi con una percentuale di superamento pari a 80.	✓
Desiderabile	Completare l'implementazione dei microservizi con una percentuale di superamento parti a 100.	✗
Facoltativo	Studiare come poter realizzare le *best practice* dell'architettura a microservizi con Node.js.	✓

4.2 Maturazione delle conoscenze professionali

Le conoscenze personali da me maturate non si limitano al solo ambito della produzione del *software*. Queste ultime hanno comunque giocato un ruolo fondamentale nella mia maturazione personale durante l'attività di tirocinio, secondo la logica da me riportata nella sezione 4 del capitolo 2: le nozioni teoriche affrontate sono risultate essere alla mia portata proprio perchè naturale continuazione del percorso da me seguito in ambito universitario in relazione allo sviluppo di applicativi *web*, e grazie proprio a queste attività ho avuto modo di comprendere il contenuto teorico proposto senza difficoltà significative ed implementare le funzionlità richieste con cognizione di causa. Detto ciò, le nozioni affrontate hanno comunque richiesto uno sforzo non banale per essere comprese correttamente e in particolare le modalità con cui si è svolto lo studio (non essendo moderato da una figura come quella del Professore che guida uno studente e filtra i contenuti proposti) sono risultate impegnative, senza però mai scadere nel tedioso o nel sovraccarico di lavoro.
Allo stesso modo, doversi confrontare con metodologie di collaborazione e di comunicazione che non sono più solamente simulative di una reale attività di sviluppo ma sono esse stesse parte di un *modus operandi* proprio di un reale contesto aziendale ha contribuito a migliorare le nozioni teoriche apprese durante il corso di studi e dare più concretezza a molte di queste che, affrontate solamente in quanto tali, risultavano essere vuote e non stimolavano una reale riflessione.
Cercando di essere più specifico rispetto alle conoscenze maturate, elenco di seguito quelle più significative.

Architettura a microservizi: sebbene fosse un tema già trattato durante il corso di Ingegneria del *Software*, durante lo svolgersi dell'attività di tirocinio ho avuto modo di rivedere e migliorare la mia compresione rispetto a questo tema e di studiare in particolare le principali componenti che permettono di implementare questo stile architetturale. Ritengo inoltre che uno studio serio e approfondito di questo tema, molto attuale, non possa prescindere da un attività di carattere più pratico che in questo contesto si è rivelata essere un'intuizione vincente;

Linguaggio Typescript e ambiente Node.js: come per l'architettura a microservizi, il linguaggio Typescript e il *runtime* Node.js erano già stati affrontati durante il corso di studi, questa volta però non come materiale effettivamente trattato durante le lezione ma come parte delle conoscenze acquisite autonomamente per affrontare attività di progetto. In particolare, lo sviluppo mediato dal *framework* Nest.js ha contribuito in maniera significativa a migliorare la mia comprensione circa i meccanismi che regolano la configurazione di un progetto per mezzo di Node.js, tema su cui sentivo di essere ancora carente e i cui meccanismi mi erano ancora oscuri;

Strumenti di comunicazione e collaborazione: durante l'attività di tirocinio ho avuto modo di sperimentare con mano molti degli strumenti di comunicazione e collaborazione che l'azienda utilizza internamente (*Discord*, *Trello*, *Google Calendar*, *Git e GitHub*). Tutti questi *software* erano a me già noti grazie ad alcuni corsi universitari e ad approfondimenti svolti in autonomia, ma prima di questa attività non ho mai avuto occasione nè di sfruttarli in maniera ragionevolmente approfondita nè di constatare quanto l'utilizzo combinato di questi possa giovare all'ambiente di lavoro. Detto ciò, sento che è necessario sottolinare anche che si tratta sì di strumenti potenti e che agevolano lo sviluppo, ma solamente nel caso in cui questi vengano utilizzati in maniera

virtuosa. Se percepiti come inutili e utilizzati senza cognizione di causa, questi possono appesantire il carico di lavoro e rendere le attività che devono moderare ancora più caotiche e difficile da gestire;

Gestione autonoma del lavoro: uno degli obiettivi fissati dall'azienda era di riuscire a implementare i servizi proposti in autonomia e rispettando il cronoprogramma. Storicamente, durante il mio percorso di studi universitario (ma analogamente anche per gli studi precedenti), ho sempre trovato molto difficile equilibrare il tempo che dedicavo alle attività che affrontavo, in alcune occasioni fallendo e trovando molto difficile valutare le ragioni dietro a tali fallimenti. Quando ho iniziato l'attività di tirocinio temevo di cadere ancora una volta in questo tranello, non riuscendo ad affrontare nella maniera corretta l'organizzazione del lavoro. Tale circostanza non si è verificata, e sono riuscito a produrre quanto richiesto e a raggiungere gli obiettivi prefissati. In questo senso, un utilizzo ben regolato e critico degli strumenti di comunicazione e collaborazione, unito al costante supporto fornitomi dal *tutor* interno, hanno giocato un ruolo fondamentale nel permettermi di svolgere tutte le attività nei tempi e modi previsti.

4.3 Aspettative soddisfatte e valutazione personale

La scelta di affrontare l'attività di tirocinio presso SyncLab è stata guidata dalla volontà di partecipare ad una attività che mi permettesse utilizzare le conoscenze maturate in ambito universitario calandole però in un contesto aziendale e applicandole a tematiche all'avanguardia nel panorama tecnologico contemporaneo. Tale aspettativa non è stata disattesa, e anzi è stata valorizzata da aspetti collaterali dell'attività da me non preventivati e che si sono in realtà rivelati essere di grande valore (si veda la sezione precedente). Il mio *tutor* interno ha dimostrato una grande competenza professionale, e da parte sua ho ricevuto un supporto attivo e costante sia dal punto di vista operativo che umano. L'ambiente in cui sono stato inserito si è rivelato essere accogliente e collaborativo. Ci sono state numerose opportunità di confronto con i dipendenti dell'azienda, e in nessun caso durante le interazioni avute ho percepito da parte loro fastidio o disinteresse nei miei confronti.
Di contro, le interazioni appena descritte si sono limitate a momenti circoscritti dell'esperienza a causa delle dinamiche proprie dello *smart working*. Mi sono recato fisicamente nella sede di riferimento meno di quanto mi aspettassi, limitando quindi le occasioni per confronti e scambi di idee.
Per quanto riguarda invece la preparazione fornitami dal corso universitario, questa si è rivelata adeguata dal punto di vista metodologico e di *forma mentis*. È solo grazie ad essi che sono riuscito ad affrontare i contenuti proposti senza mai sentirmi impreparato anche quando i contenuti proposti mi erano completamente sconosciuti. La principale critica che muovo invece riguarda la distanza tra i concetti teorici fornitimi nel tempo e le capacità richieste dal punto di vista operativo: spesso durante il mio percorso di studi ho avuto la sensazione di star studiando concetti teorici che non venivano opportunamente calati in un contesto operativo. Questo non si limita ai corsi che prevedono attività di laboratorio o progettuali, ma anche nei corsi di stampo puramente teorico.
Ritengo sia bene, per avere una lettura contestualizzata di quanto appena detto, fornire un dato ulteriore riguardo la mia persona: prima di iniziare il mio percorso

universitario non ho mai avuto modo di cimentarmi in attività di codifica o *test* di prodotti *software*, e il mondo della *computer science* e delle sue pratiche erano molto distanti dalle mie conoscenze e competenze. Le nozioni di natura teorica hanno dovuto pertanto essere calate a livello applicativo spesso richiedendo che fossero studiate e interiorizzate pratiche ulteriori non affrontate dai corsi proposti. Un'altra critica che mi sento di muovere è quella per cui in alcuni corsi i contenuti di carattere tecnologico sono risultati essere obsoleti rispetto al panorama contemporaneo, portando alle medesime conseguenze del fenomeno prima descritto, ovvero la necessità di integrare in autonomia i materiali forniti. Gli aspetti appena riportati non sono comunque sufficienti a invalidare la qualità dell'offerta formativa proposta: prima di iniziare il mio percorso di studi sarebbe per me stato impossibile anche solo pensare di avvicinarmi al mondo della produzione del *software*, ed ora ciò non è più vero e la sola esistenza di questa relazione ne è una chiara dimostazione.

Appendice A

Nozioni teoriche complementari

A.1 Monolite e microservizi a confronto

Per poter comprendere appieno il lavoro che ho svolto, è necessario comprendere l'architettura con la quale vengono implementati gli applicativi prodotti. In particolare, prima di poterci soffermare sul concetto di microservizio è necessario capire perchè questo si è reso necessario e quindi i problemi che cerca di risolvere. Assumiamo di iniziare lo sviluppo di un nuovo applicativo. Dopo le fasi preliminari, verrebbe generato un nuovo progetto strutturato utilizzando, ad esempio, il *pattern* dell'architettura esagonale nel contesto di un applicativo monolitico. Il termine **monolite**, già utilizzato in precedenza ma mai definito con precisione, identifica nell'ambito *web* un applicativo sviluppato come singola unità coesa che raccoglie tutte le logiche necessarie al corretto funzionamento del prodotto finale. Si tratta di uno stile architetturale estremamente diffuso, considerando anche che è stato uno dei primi ad essere adottato nell'ambito dello sviluppo *software*.
La struttura prevede due "versanti" principali:

- **Core**: centro dell'architettura, implementa servizi, oggetti di dominio e eventi. In altre parole, tutto ciò che riguarda effettivo *processing* dei dati, e non la comunicazione. Nel grafico sopra riportato è rappresentato con l'esagono centrale;
- **Adapters**: parte più esterna dell'architettura, che si compone di tutta una serie di *adapters* che permettono la comunicazione tra il servizio centrale e quelli esterni.

Logicamente la struttura presentata è modulare, ma solo logicamente. Di fatto, il *packaging* e *deployment* dell'applicativo viene fatto "monoliticamente". Ciò può portare a scenari indesiderati in contesti differenti:

- **Codifica**: modifiche al codice sorgente rischiano di mettere in crisi l'integrità del lavoro svolto (regressione) e l'aumento di volume della *codebase* può portare ad uno scenario in cui non è più possibile comprendere intuitivamente i meccanismi che regolano l'applicativo nella sua interezza;
- **Gestione dello sviluppo**: per loro natura, lo sviluppo di applicativi monolitici è antitetico alle metodologie *agile*;
- ***Deploy***: i tempi di *startup* possono diventare significativi all'avvio di una nuova istanza del *software*, e applicativi che svolgono funzioni eterogenee potrebbero

richiedere lo sviluppo di sistemi con *hardware* specifico (con costi di progettazione e realizzazione associati);

- **Afffidabilità**: un problema riscontrato su anche solo uno dei servizi offerti da un applicativo di tipo monolitico potrebbe paralizzare l'intero sistema, andando quindi a bloccare anche la corretta esecuzione di quelle funzionalità che non sono interessate da alcuna anomali.

Tale *pattern* può essere rappresentato come nel diagramma che segue:

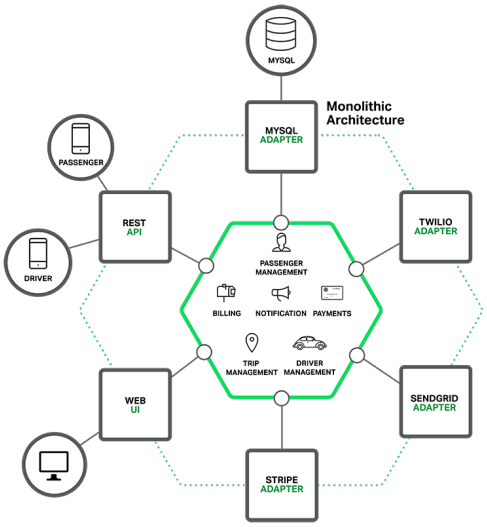


Figura A.1: Rappresentazione visiva di un applicativo monolitico strutturato secondo lo schema proprio di un architettura esagonale.
Fonte: nginx.com

Quando queste problematiche si manifestano gli sviluppatori tendono a riferire il risultato dello sviluppo con lo scherzoso ma calzante appellativo di *"Big ball of mud"* (letteralmente, "grande palla/ammasso di fango") per enfatizzarne la natura caotica e degenerativa dello sviluppo.
Una possibile soluzione consiste nell'adozione del "*Pattern* Architetturale a Microservizi" (nelle fonti da me consultate identificati con il termine "Microservices architectural *pattern*"). L'intuizione è quella di dividere l'applicativo in tante applicazioni più piccole, interconnesse tra loro. Un servizio tipicamente implementa un sottoinsieme specifico di funzionalità, e possiede una propria architettura simile a quella vista in precedenza (che comprende quindi una *business logic* per l'elaborazione dei dati e svariati *adapters* per la gestione della comunicazione). Dal punto di vista della scalabilità, l'architettura a micro-servizi lavora molto sulla decomposizione funzionale (le prestazioni dell'applicativo vengono scalate attraverso una divisione nelle sue diverse componenti). Ciò non significa che altre tipologie di *scaling* differenti, ma l'architettura riportata implementa nello specfico la tipologia riportata. E' inoltre fondamentale osservare che ciascun microservizio dispone di un proprio *database* (*database schema*) con la quale interagisce autonomamente e slegandosi completamente dagli altri servizi.

Quella rappresentata di seguito è l'architettura monolitica vista inizialmente, ma trasposta a microservizi.

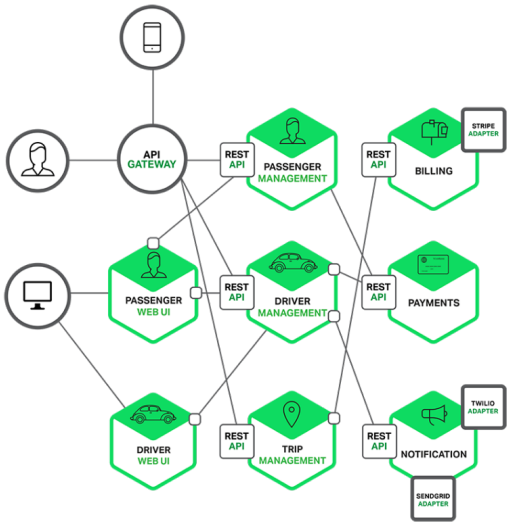


Figura A.2: Rappresentazione visiva di un applicativo a microservizi strutturato secondo lo schema proprio di un architettura esagonale. Si tratta della medesima architettura vista in precedenza, ma trasposta a microservizi. Si può osservare come l'architettura esagonale in questo caso viene applicata ai singoli servizi, in questo contesto rappresentati appunto mediante una forma ad esagono.
Fonte: nginx.com

I principali vantaggi nell'utilizzo dei microservizi rispetto al monolite sono:

- **Riduzione della complessità**: il monolite e i microservizi sono perfettamente equivalenti dal punto di vista funzionale. Tutto ciò che uno può, può anche l'altro. Ma le dimensioni dei microservizi, essendo incredibilmente ridotte, permettono una gestione molto più agevole della *codebase* (aggiornamento, espansione, manutenzione, modularità);

- **Incapsulamento logico e sviluppo specifico**: lo sviluppo dei singoli microservizi è indipendente dal funzionamento degli altri. Ciò si traduce nella possibilità per gli sviluppatori di concentrarsi sullo sviluppo di un servizio alla volta, indipendentemente dalla struttura logica complessiva e in totale isolamento (a patto di offrire un *API* come interfaccia di interazione);

- **Aggiornamento di tecnologie**: nuovi servizi non necessitano di utilizzare sempre le stesse tecnologie, ma alla creazione di un nuovo servizio si può scegliere la migliore disponibile in quel momento e, nel caso si decida di aggiornare un servizio che usa tecnologie datate, l'onere diminuisce significativamente;

- ***Deploy* indipendente**: ogni servizio può essere istanziato indipendentemente, e diventa possibile fare *Continuos Deployment* (quindi, fare *deploy* non appena si passano i *test* in maniera automatizzata);

- ***Scaling* indipendente dei servizi**: ogni servizio può scalare in maniera indipendente dagli altri. E, nel caso in cui un determinato *hardware* sia desiderabile al fine di performare determinate operazioni, è possibile fare *deploy* su macchine con caratteristiche diverse, che meglio si adattano alle esigenze del singolo servizio.

A queste si accompagnano ovviamente anche annessi svantaggi:

- **Il termine microservizio è fuorviante**: la natura contenuta del singolo microservizio non è perentoria, e risulta essere virtuosa solo nel momento in cui la complessità del servizio implementato lo permette. Perseguire acriticamente tale obiettivo può compromettere lo sviluppo e rischia di complicare inutilmente lo sviluppo;
- **La complessità di configurazione aumenta significativamente**: se per un applicativo monolitico è necessario configurare uno ed un solo applicativo, nel contesto a microservizi tale onere si diffonde "a macchia d'olio" su ciascuno dei servizi coinvolti. A ciò, vanno aggiunti anche gli oneri implementativi rispetto ai meccanismi che regolano la comunicazione tra i servizi.

La complessità è stata quindi spostata dal singolo applicativo ai meccanismi di comunicazione e moderazione dell'interno applicativo (risultato dell'azione coordinata dei singoli servizi), e come detto si rende necessaria la definizione di componenti *software* adatte allo scopo. Tali componenti sono esaminate ed esposte nella sezione successiva.

A.2 Componenti architetturali di un infrastruttura a microservizi

Come già accennato nella sezione precedente, il passaggio da un architettura di tipo monolitico ad una a microservizi richiede l'implementazione di componenti *software* aggiuntive per la moderazione del sistema e la gestione del flusso di informazioni. Trattandosi di un architettura distribuita risulta necessario definire meccanismi di controllo del carico di lavoro, di gestione di risposte tardive/mancanti/non conformi e un sistema di messaggistica tra le varie componenti. Durante la mia attività di tirocinio, in particolare prima di iniziare lo studio vero e proprio dei *framework* presi in esame, mi sono concentrato sullo studio teorico delle componenti fondamentali per questo stile architetturale. Tale insieme comprende:

- *API Gateway*;
- *Service Registry e Service Discovery*;
- *Circuit breaker*;
- *Saga pattern*;

Di seguito tratterò le singole tecnologie esaminandone il raziocinio alla base nel contesto dei microservizi (nel tentativo di rispondere alla domanda "perchè questo componente è necessario?").

API Gateway: In un contesto di *API REST*, non è impossibile incontrare scenari in cui bisogna fornire diverse tipologie di dati a endpoint di natura molto diversa

tra loro. Ad esempio, i dati restituiti ad una applicazione Android / iOS saranno molto diversi dai dati forniti ad uno stesso servizio che però viene acceduto da una piattaforma come un Web Browser. Nel caso poi di un architettura a microservizi diventa fondamentale capire come implementare la comunicazione tra client e i singoli servizi. L'approccio proposto per risolvere questi problemi prevede l'utilizzo di quello che viene chiamato *API Gateway*. Per dare più concretezza a quanto appena detto propongo un esempio che parte dall'analisi di una pagina per un prodotto per il sito di *Amazon*. Tale pagina contiene una varietà di informazioni che devono essere recuperate dal *client*. In un applicativo monolitico, si avrebbe un unica chiamata *REST* indirizzata a un istanza attiva del servizio. In un applicativo che fa utilizzo dell'architettura a microservizi invece le singole informazioni dovrebbero essere recuperate da ciascuno dei servizi preposti (es. Numero di articoli nel carrello dal Shopping Cart Service, Oggetti consigliati da Recomendation Service(s) ecc. . .). Intuitivamente sarebbe possibile pensare di effettuare una sequenza di chiamate *REST* a cascata verso tutti i microservizi coinvolti, ma questo approccio appare immediatamente come poco efficiente e *error prone* (un elevato numero di chiamate ai servizi corrisponde contestualmente a un onere di codifica maggiore, che a sua volta introduce automaticamente la possibilità di introdurre *bug* ed errori). La figura riportata di seguito rappresenta lo scenario appena descritto, evidenziando in particolare come non vi sia alcun tipo di struttura associata all'applicativo finale.

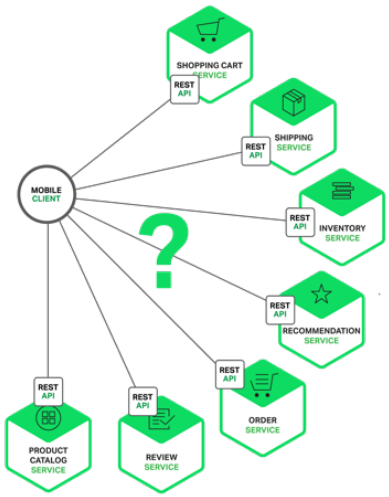


Figura A.3: Rappresentazione visiva di un applicativo a microservizi in cui le logiche di comunicazione non sono filtrate da alcune componenti e le informazioni sono recuperate e filtrate direttamente dal *client* del servizio.
Fonte: nginx.com

Per risolvere questo problema, viene introdotto un componente *software* ausiliario, ovvero un *API Gateway* che ha sostanzialmente 3 oneri fondamentali:

- Gestire il *routing* delle richieste;
- Composizione delle risposte;

- *Protocol translation*, ovvero permettere a componenti che utilizzano protocolli di comunicazione differenti di interagire senza aggiungere complessità alle componenti stesse.

Esso quindi funge da gestore delle chiamate ai singoli servizi: si occupa di ricevere tutte le chiamate in ingresso, di chiamare i singoli servizi coinvolti e comporre le risposte da fornire ai *client*. In altre parole astrae la moltitudine di servizi fornendo un unico punto di accesso a tutte le informazioni.

Service Registry e Service Discovery: nel descrivere i microservizi nella sezione precedente si è fatto riferimento al grande vantaggio che questi hanno rispetto alla controparte monolitca nel permettere al servizio di scalare in maniera efficace. In termini concreti, questo si traduce nella possibilità di istanziare (quando necessario) solamente i servizi che lo richiedono effettivamente. Tali servizi, per essere disponibili alla comunicazione dovranno necessariamente essere individuabili in una rete, e quindi essere in possesso di un indirizzo *IP* univoco. Le componenti del *Service Registry* e del *Service Discovery* servono proprio per permettere di identificare e gestire l'istanziazione di nuovi servizi rendendo il processo trasparente alle componenti che necessitano di interagire con i servizi stessi.
La figura sottostante inquadra il problema dal punto di vista visuale:

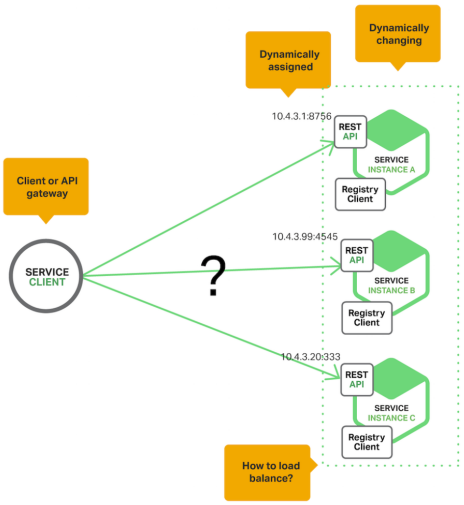


Figura A.4: Una rappresentazione visiva del problema che si vuole affrontare e risolvere per mezzo di *Service Discovery* e *Service Registry*. Come identificare l'indirizzo verso cui effettuare la richiesta minimizzando la complessità del codice che svolge tale compito?

Fonte: nginx.com

Ecco quindi che entrano in gioco le componenti del *Service Discovery* e *Service Registry*. Il primo dei due assolve all'onere di fornire le informazioni che identificano un servizio specifico tra le molte istanze disponibili, mentre il secondo funge da registro per la memorizzazione delle informazioni relative ai servizi disponibili in un dato momento.

In altre parole, la componente che assolve all'onere di *Service Discovery* si appoggia a sua volta ad un *Service Registry* per recuperare e rendere fruibili le informazioni sui servizi con la quale si desidera comunicare. Il pattern del *Service Discovery* è implementabile seguendo due metodologie differenti:

- ***Client-side Discovery***: in questo caso il *client* dialoga direttamente con un *Service Registry* e dopo aver ricevuto le informazioni relative agli indirizzi *IP* dei servizi disponibili seleziona autonomamente l'istanza migliore con cui dialogare (questo meccanismo viene riferito con il termine di *load balancing*, traducibile con l'espressione "gestione del carico");

- ***Server-side Discovery***: in questo caso tra *client* e *Service Registry* viene interposto un componente aggiuntivo che si occupa di recuperare le informazioni relative ai servizi interessati dalla chiamata e "confeziona" le informazioni in modo che siano direttamente disponibili per il *client*.

Circuit breaker: per introdurre cosa è un *Circuit Breaker* si può pensare ad un analogia con il sistema elettrico domestico. Può capitare che il flusso di corrente che scorre lungo la rete abbia un comportamento anomalo. In questi casi, si attivano dei meccanismi di protezione e prevenzione che bloccano il flusso e prevengono consegnuenze catastrofiche. Il *Circuit Breaker* ha la stessa funzione, ma applicata al contesto di micro-servizi. Serve dunque a gestire e moderare scenari in cui i servizi non sono disponibili e si vuole garantire integrità al funzionamento del macro-servizio che si appoggia a servizi individuali. Nel caso delle architetture a microservizi, non è raro incontrare scenari in cui un particolare servizio non è disponibile per tutta una serie di motivi (lentezza nella connessione, timeouts o temporanea indisponibilità). Queste casistiche poi possono portare a errori a cascata su tutta l'applicazione. Onde evitare situazioni di questo tipo è possibile utilizzare un componente, il *Circuit Breaker* appunto, che funge da *proxy* per le chiamate dirette verso un servizio. Se il numero di chiamate non andate a buon fine supera una certa soglia, il *proxy* blocca le connessioni a quel particolare servizio per un periodo di tempo limitato e ogni tentativo di connessione viene fatto fallire. Allo scadere del tempo di *timeout*, il *proxy* accoglie un numero limitato di richieste per testare se il servizio ha ripreso il corretto funzionamento. In caso di esito negativo, il timeout viene fatto ripartire.

Saga pattern: Uno dei problemi principali nei sistemi distribuiti è la gestione della coerenza (*consistency*) dei dati del loro flusso. È desiderabile che in applicativi dove la manipolazione dei dati è massiva e ad opera di servizi diversi venga garantita suddetta coerenza e affidabilità dei dati stessi e che in caso di errore vi sia un meccanismo di recupero dello stato precedente all'errore. Una modalità attraverso cui è possibile gestire suddetti scenari è il *Saga Pattern*. L'idea alla base del Saga Pattern consiste nella gestione di transazioni distribuite ognuna delle quali rappresenta, appunto, una saga. Il *focus* principale è garantire il corretto ordine di esecuzione delle operazioni che fanno parte della stessa saga, ovvero che devono essere eseguite in un ordine prestabilito e il cui effetto non è compromesso da eventuali operazioni intermedie appartenenti a saghe diverse. Questo pattern aiuta quindi a gestire la consistenza dei dati nell'esecuzione di transazioni distribuite tra microservizi diversi. L'obiettivo è duplice: mantenere l'identità del dato ed effettuare azioni di compensazione per il ripristino in caso di errore. Esistono due approcci differenti all'implementazione di questo *pattern*:

- **Events / Coreography**: in questo caso non vi sono coordinatori (ovvero componenti *software* intermedie), i servizi lavorano insieme ma senza essere supervisionati per completare la saga in esecuzione;
- **Commands / Orchestration**: in questo caso il controllo viene centralizzato e gestito da un singolo componente chiamato in questo contesto orchestratore. Lo svolgersi di una saga è quindi supervisionato e non più delegato alla sola implementazione dei meccanismi di moderazione per mezzo dei singoli micro-servizi;

Come già detto più volte, le componenti appena esposte non state implementate (in quanto non richieste e realisticamente troppo onerose) durante l'attività di tirocinio da me svolta. Ciò nonostante, gli applicativi prodotti sono stati progettati avendo come orizzonte un loro utilizzo in un contesto che utilizza e implementa le componenti descritte in sopra.

Bibliografia

Risorse documentali consultate

- ***Introduction to microservices***, di *Chris Richardson*, 2015. URL: https://www.nginx.com/blog/introduction-to-microservices/;
- ***Building microservices: Using an API Gateway***, di *Chris Richardson*, 2015. URL: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/;
- ***Service Discovery in a microservice Architecture***, di *Chris Richardson*, 2015. URL: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/;
- ***Design Patterns for Microservices — Circuit Breaker Pattern***, di *Nisal Pubudu*, 2021. URL: https://medium.com/nerd-for-tech/design-patterns-for-microservices-circuit-breaker-pattern-ba402a45aac2;
- ***Introduction to Spring Data JPA***, di *Eugen Paraschiv*, 2022. URL: https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa;
- **Documentazione Java Spring**, di *SpringSource*, 2022. URL: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html;
- ***Building a RESTful Web Service***, di *SpringSource*, 2022. URL: https://spring.io/guides/gs/rest-service/;
- **Documentazione NestJS**, 2022. URL: https://docs.nestjs.com/;
- ***Getting Started with NestJS***, di *Olususi Kayode Oluyemi*, 2022. IRL: https://www.digitalocean.com/community/tutorials/getting-started-with-nestjs.