ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

DEVOPS E SOFTWARE CONTAINERS IN ARCHITETTURE A MICROSERVIZI

Elaborato in
SISTEMI EMBEDDED E
INTERNET OF THINGS

Relatore
Prof. ALESSANDRO RICCI

Presentata da
FRANCESCO DENTE

Corelatore
Dott. Ing. ANGELO CROATTI

Anno Accademico 2018 – 2019

Indice

Introduzione

Il mercato informatico odierno ha visto una crescita esponenziale sotto svariati punti di vista. Da un lato, la proliferazione delle software house, sia di piccole, sia di grandi dimensioni, ha fatto sì che la pressione per restare competitivi sul mercato sia diventata una problematica non più trascurabile dai responsabili aziendali. Dall'altro lato, la crescente complessità delle soluzioni informatiche richieste al giorno d'oggi inizia a rendere proibitive le metodologie usate tradizionalmente nello sviluppo di sistemi, richiedendo quindi una nuova mentalità nell'affrontare la gestione dei progetti.

È da queste necessità che nasce il movimento DevOps, che si impone come un nuovo modo per affrontare il ciclo di vita di un progetto allo scopo di velocizzarne i tempi di produzione, assicurando allo stesso tempo un alto livello di qualità. Le sfide che DevOps richiede di affrontare non hanno ovviamente un costo nullo, e prevedono al contrario una pesante ristrutturazione in termini *architetturali*, *tecnologici* e *organizzativi*.

Il movimento introduce infatti un nuovo stile architetturale, i *microservizi*, con il quale si decompone la complessità del dominio applicativo in varie componenti software distribuite che vengono affidate ai singoli team di sviluppatori incoraggiando uno sviluppo più agile e indipendente. Mentre essi semplificano gran parte dello sviluppo, generano allo stesso tempo delle nuove sfide che riguardano aspetti più trasversali rispetto alla semplice programmazione, che richiedono nella maggior parte dei casi tecnologie o framework specifici. Tra queste, fondamentali sono la necessità di avere un deployment consistente e standardizzato in combinazione con un monitoraggio efficace, che in un ecosistema distribuito e replicato possono risultare troppo delicati per essere gestiti manualmente. In questo frangente brillano le recenti tecnologie di *containerizzazione del software* e di *orchestrazione di container*, pensate proprio per assorbire la complessità della distribuzione dei sistemi.

Nel corso di questa trattazione, si vuole proporre una panoramica dei principi fondamentali della cultura DevOps, introducendo quindi le motivazioni che hanno portato alla nascita dei microservizi come modello architetturale di riferimento e le modalità con cui Docker e Kubernetes – usati rispettivamente come strumenti di containerizzazione e orchestrazione – risolvono alcune

problematiche che sorgono dalla loro adozione.

Lo studio effettuato verrà infine applicato a un progetto già in fase di sperimentazione nell'ospedale Bufalini di Cesena a supporto dei medici del trauma team: il sistema Trauma Tracker. Il sistema, che già al momento segue lo stile dei microservizi, è in uno stadio iniziale del suo sviluppo e va incontro a una lunga serie di aggiornamenti che aggiungeranno nuove funzionalità, ma allo stesso tempo ne aumenteranno la complessità e l'organizzazione. La scelta di attribuire all'ecosistema Trauma Tracker un'architettura più modulare farà sì che la crescita funzionale del progetto abbia un costo relativamente basso. Tuttavia, la mancanza di una standardizzazione robusta nella messa in attività del sistema sta rendendo troppo fragile l'effettiva operatività del sistema, che è di tanto in tanto soggetto a malfunzionamenti difficilmente individuabili o riproducibili dagli sviluppatori.

L'obiettivo che si pone il lavoro di questa tesi è quindi quello di attribuire a Trauma Tracker una nuova modalità di deployment, che sfrutta i container Docker per impacchettare le applicazioni in unità standard e pronte per il deployment, unita alla potenza di Kubernetes per orchestrare facilmente l'esecuzione e l'interconnessione tra servizi, occupandosi automaticamente di eventuali guasti o malfunzionamenti dell'infrastruttura fisica. Si cercherà quindi di avvicinare il più possibile l'organizzazione e l'architettura del progetto agli standard richiesti da DevOps, per fare in modo che il futuro del progetto ponga un numero minimo di ostacoli di natura operativa agli sviluppatori, consentendo loro di concentrarsi maggiormente su aspetti prettamente progettuali e implementativi.

Capitolo 1

DevOps

Quando si parla di DevOps si fa riferimento a un movimento culturale che punta a migliorare la collaborazione e la coordinazione tra sviluppatori (Dev, Developers) e addetti alle operazioni (Ops, Operations), ovvero i responsabili di installazione e manutenzione del software. Bass, Weber e Zhu definiscono DevOps come *un insieme di pratiche che hanno lo scopo di ridurre il tempo tra il commit di un cambiamento a un sistema e l'effettiva messa in produzione del cambiamento, assicurando nel frattempo un alto livello di qualità* [2]. Entrambe le definizioni rispecchiano l'essenza di DevOps, ma differiscono per il concetto su cui è posto il focus. Mentre la prima si concentra maggiormente sulla metodologia da mettere in atto, la seconda mette al centro dell'attenzione lo scopo finale di DevOps: velocizzare la messa in produzione di un sistema.

1.1 Modelli di sviluppo software

Per *modello di sviluppo* si intende una suddivisione del ciclo di vita del software in più fasi al fine di migliorarne il design, lo sviluppo e la manutenibilità. Ogni fase produce un certo insieme di *artefatti*, anche detti *deliverables*, che vengono spesso utilizzati come punto di partenza per le fasi successive. Due modelli rilevanti al fine di introdurre le motivazioni dietro DevOps sono i *modelli a cascata* e i *modelli agili*, descritti di seguito.

1.1.1 Modello a cascata

Il modello a cascata rappresenta uno dei primi modelli di sviluppo della storia dell'IT (1970). Esso divide il ciclo di vita in cinque passi distinti e prevede il passaggio in cascata degli artefatti prodotti da ciascuna fase alle fasi successive, senza la possibilità di tornare a fasi già concluse. Più nello specifico si individuano le seguenti fasi (Figura 1.1):

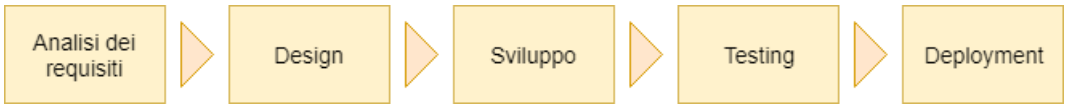


Figura 1.1: Rappresentazione schematica del modello a cascata

- **Analisi dei requisiti**: si intervista il cliente per capire quali sono le funzionalità che dovrà avere l'applicazione. Il risultato di questa fase è un documento che contiene la *specifica dei requisiti*.
- **Design**: partendo dalla specifica dei requisiti si definisce un *piano di progetto* e il *design architetturale* utilizzato dagli sviluppatori nella fase successiva.
- **Sviluppo**: il software viene creato secondo lo schema architetturale stabilito.
- **Testing**: si eseguono test di vario genere per assicurare che il prodotto sia privo di errori prima della consegna al cliente.
- **Deployment**: il software viene messo in produzione ed è compito dell'azienda produttrice mantenerlo in funzione.

La rigidità del modello a cascata crea delle problematiche non trascurabili. Nel mondo informatico odierno è infatti molto raro avere requisiti stabili per tutta la durata del progetto. La possibilità che il cliente si presenti con nuovi requisiti, anche in fasi avanzate del progetto, richiede nella maggior parte dei casi di far ripartire completamente il ciclo di sviluppo, naturalmente con elevati costi per l'azienda. Inoltre il modello a cascata prevede che il cliente riceva il prodotto solo una volta completato l'intero ciclo. Se dopo la consegna si scoprissero funzionalità implementate in maniera non corretta o che più semplicemente non soddisfano il cliente, ancora una volta si dovrebbero ripetere tutte le fasi dall'inizio.

La conclusione che si può facilmente trarre è che la mancanza di chiarezza nei requisiti comporta, nella maggior parte dei casi, un significativo ritardo nella consegna del prodotto e un costo non indifferente. Mentre in genere sia molto complicato rendere stabili i requisiti, è possibile se non altro adottare un modello che favorisca maggiormente la comunicazione con il cliente e che sia più adatto a far fronte ai cambiamenti frequenti.

1.1.2 Modello agile

Con l'uso del modello a cascata ci si è resi conto che il mercato informatico moderno richiede che le soluzioni siano consegnate molto più velocemente e

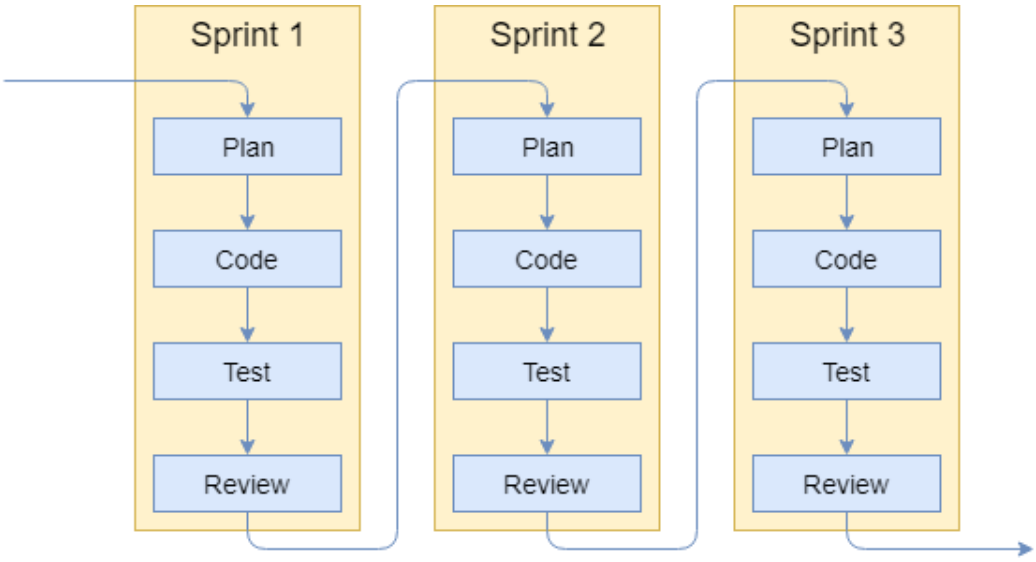


Figura 1.2: Rappresentazione schematica del modello agile

che ci sia un continuo feedback da parte del cliente. Il *modello agile* nasce proprio con questo obiettivo.

Seguendo metodologie agili gli sviluppatori creano periodicamente dei prototipi che hanno un duplice scopo: rendere il cliente consapevole dell'avanzamento del progetto, ma soprattutto verificare la comprensione dei requisiti. Si vengono così a creare dei cicli di confronto tra cliente e azienda che hanno una durata molto breve (in genere 2-3 settimane) e durante i quali si fa riferimento al feedback del cliente. Questi cicli prendono il nome di *sprint* (Figura 1.2).

Ogni sprint si divide approssimativamente in quattro fasi:

- **Plan**: si decide, con l'intervento del cliente, quali funzionalità implementare durante lo sprint.
- **Code**: si implementano le funzionalità stabilite, avvalendosi della prototipazione in caso di specifiche poco chiare.
- **Test**: si verifica che ciò che è stato implementato sia privo di errori, spesso facendo uso di test automatici.
- **Review**: il cliente fornisce il suo parere sul lavoro svolto, indicando problematiche riscontrate che possono essere risolte per il prossimo sprint.

Nonostante l'impiego di metodologie agili rappresenti un netto miglioramento, alcune problematiche sono ancora presenti. Se il prodotto non viene testato nell'ambiente in cui si troverà in produzione (o in uno sufficientemente

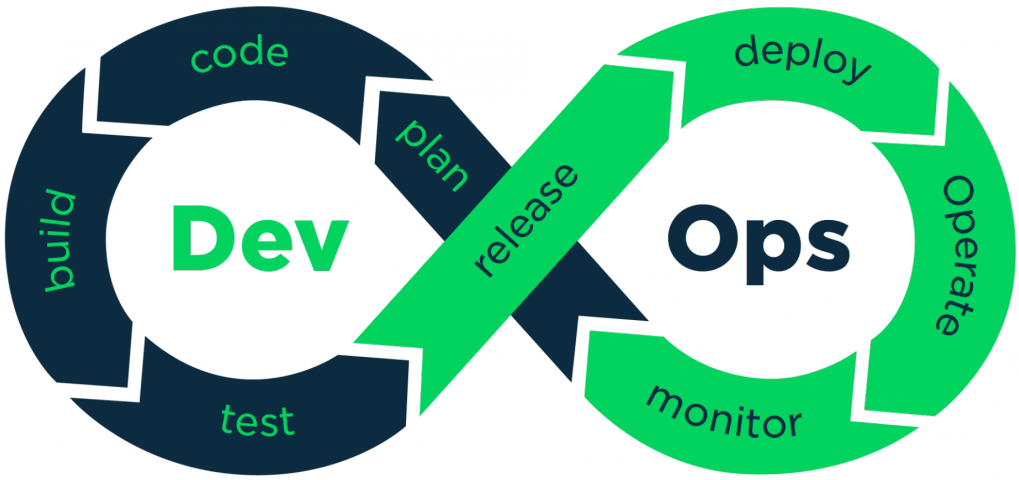


Figura 1.3: Rappresentazione del modello di sviluppo proposto da DevOps

simile), ma solo sulle macchine degli sviluppatori, è molto probabile che una volta installato ci siano errori non riscontrati prima. In aggiunta il team degli sviluppatori e quello degli operatori lavorano senza nessun tipo di collaborazione. Una volta terminato lo sprint, il software viene semplicemente messo nelle mani dei sistemisti ed è compito loro trovare il modo di farlo funzionare stabilmente nell'ambiente di destinazione. Naturalmente questo compito non è semplice: l'ambiente di produzione potrebbe avere diverse configurazioni, librerie mancanti, software di terze parti non installato o ancora avere un carico di lavoro molto superiore. Il team di operatori si trova quindi a far fronte a una grossa sfida: risolvere problemi senza avere un'adeguata conoscenza del codice.

Si rende quindi necessaria la collaborazione tra i due mondi per rendere più agevole la messa in produzione. Si gettano così le basi per la cultura DevOps e il suo modello di sviluppo.

1.2 Il modello adottato da DevOps

DevOps propone un nuovo modello che si basa in buona parte su quello agile, estendendone le caratteristiche. Come mostrato in Figura 1.3 sono presenti otto stadi. I primi quattro descrivono azioni storicamente del team Dev:

- **Plan**: pianificazione delle prossime feature da implementare.

- **Code**: sviluppo delle nuove feature, facendo uso di tool per il *version control*.
- **Build**: compilazione consistente delle singole componenti.
- **Test**: testing delle varie componenti usando tool automatici e non.

I restanti quattro descrivono invece operazioni tipiche del team Ops:

- **Release**: integrazione delle componenti sviluppate dai singoli team.
- **Deploy**: installazione dell'intero sistema nell'ambiente di produzione.
- **Operate**: configurazione dell'ambiente di produzione per fare in modo che il prodotto esegua correttamente.
- **Monitor**: supervisione dell'ambiente di produzione per verificare l'impatto sull'utente finale e la presenza di errori.

Mentre col modello agile la cadenza delle release è dell'ordine delle settimane (la durata dello sprint, in sostanza), DevOps prevede che le modifiche al software siano rilasciate *più volte al giorno*. Con cambiamenti più piccoli è infatti meno probabile che ci siano errori e qualora ci fossero è significativamente più semplice individuarne la causa e porvi rimedio. La maggiore frequenza delle release rende il feedback del cliente ancora più mirato, e permette, a differenza del modello agile, di rendere più efficiente anche l'infrastruttura su cui viene messo in esecuzione il prodotto e soprattutto il processo che porta al rilascio.

Per avere questa possibilità è però fondamentale che i due team (Dev e Ops) lavorino a stretto contatto, ed è altrettanto fondamentale fare affidamento a diversi strumenti che rendono *automatica* la maggior parte delle fasi descritte precedentemente.

1.3 Pratiche della cultura DevOps

Oltre al nuovo modello, DevOps propone un insieme di pratiche che mirano a velocizzare o agevolare i processi aziendali. L'adozione di queste linee guida è indispensabile per ottenere i benefici promessi da DevOps.

1.3.1 Collaborazione di Dev e Ops

Prima dell'avvento di DevOps, il team Dev aveva un'unica responsabilità: sviluppare il codice dell'applicazione. Nel momento in cui era richiesto il rilascio di una nuova versione, l'intera code base veniva messa nelle mani del team

Ops per essere installata nell'ambiente di produzione. Dovendo far fronte a problemi di vario genere, era quindi necessario che gli operatori avessero una vaga conoscenza di come era stato implementato il codice. Gran parte del tempo veniva quindi spesa per il passaggio della conoscenza dagli sviluppatori agli operatori, con la conseguenza di rallentare la messa in produzione.

Lo scopo di DevOps è di rendere gli sviluppatori più responsabili del processo di deployment e di troubleshooting, creando il software in modo che sia facilmente monitorabile e di gestire eventuali errori sulla base del feedback fornito dal team di operatori. DevOps suggerisce che gran parte della coordinazione venga in qualche modo automatizzata o che si adottino tecniche che la rendono quanto più possibile superflua.

1.3.2 Struttura e coordinazione dei team di sviluppatori

Se l'obiettivo di DevOps è rendere più veloci e agevoli i processi di sviluppo, è necessaria un'organizzazione dei team che sia orientata a soddisfare tale obiettivo.

I team sono in genere di dimensioni piccole, in modo da favorire la velocità nel prendere decisioni e la coesione tra i membri. Questo genera lo svantaggio di dover necessariamente dividere task di dimensione troppo elevata in più task di grandezza adeguata e assegnare ciascuno a un team diverso, facendo in modo che le varie parti funzionino bene nel complesso e minimizzando il lavoro duplicato. Per rendere efficace questa operazione ci deve essere un buon grado di coordinazione tra i team. Con questo non si vuole intendere una coordinazione frequente. Al contrario, DevOps vuole minimizzare lo spreco di tempo dovuto alle continue comunicazioni tra team. A tal proposito si introducono meccanismi di coordinazione all'interno degli strumenti già in uso (per esempio, il controllo di versione) e si fa in modo che l'architettura generale del sistema sia tale da necessitare comunicazione minima. Da questa filosofia nasce il concetto di *architettura a microservizi*, descritta in dettaglio nel Capitolo 2.

All'interno del team si individuano diversi ruoli, che possono essere condivisi da più persone e anche essere sovrapposti. I principali sono:

- **Team lead**: possiede alcune abilità di project management e dirige il lavoro del team.
- **Team member o developer**: responsabile della creazione del sistema, ricopre attività come la modellazione, lo sviluppo e il testing.
- **Service owner**: responsabile per la coordinazione con gli altri team.

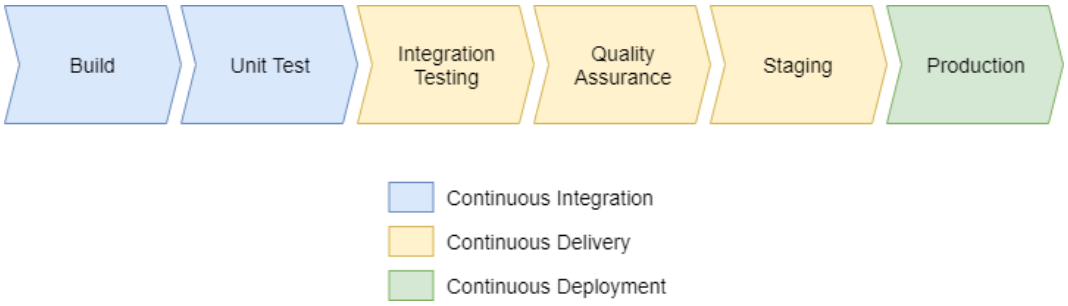


Figura 1.4: Un esempio di deployment pipeline

- **Reliability engineer**: responsabile del monitoraggio del sistema in produzione e del riscontro degli errori.

1.3.3 Continuous Integration/Continuous Delivery

Per avere la possibilità di eseguire più release in un giorno è necessario avere a disposizione un meccanismo che permetta, in tempi brevi e con un alto livello di confidenza, di decidere se l'aggiunta o la modifica di una funzionalità siano pronte per essere messe in produzione. Questa verifica viene effettuata dalle cosiddette *Continuous Integration* e *Continuous Delivery* (CI/CD).

Prima di essere pronta al deployment, ogni nuova versione deve attraversare una serie di ambienti di test chiamata *deployment pipeline*. Ad ogni stadio, la versione viene testata in un ambiente sempre più simile all'ambiente di produzione, per verificarne gradualmente il grado di correttezza. Solo una volta che tutti i test sono superati, la nuova versione può essere considerata valida.

La figura 1.4 mostra un esempio di deployment pipeline. Si noti che ogni azienda è libera di scegliere come comporre la propria pipeline a seconda delle necessità richieste dal progetto.

Ambienti Fino ad ora si è parlato di ambienti senza darne una definizione precisa. Per ambiente si intende un insieme di risorse computazionali sufficienti a eseguire un sistema software, inclusi tutti i software di supporto, i dati, le comunicazioni di rete, le configurazioni e le entità esterne necessarie a eseguire il sistema stesso [2]. Ciascun ambiente è tipicamente separato dagli altri e non condivide risorse con essi, se non quelle in sola lettura. Quest'ultima caratteristica è fondamentale ai fini della CI/CD. Se, ad esempio, l'ambiente in cui l'applicazione viene testata condividesse il database con l'ambiente di produzione, si potrebbero modificare dati reali dell'applicazione, compromettendoli.

L'astrazione di ambiente consente di avere configurazioni diverse nei vari stage della pipeline. Per esempio, l'ambiente in cui ciascuno sviluppatore testa l'applicazione prima di eseguire un commit utilizzerà un livello di logging molto più verboso rispetto a quello di produzione, per facilitare la ricerca e la correzione di bug. Nonostante questa flessibilità di configurazione sia di grande aiuto, abusarne potrebbe rendere gli ambienti di test troppo differenti rispetto a quello di produzione e influire sul comportamento del sistema.

Continuous Integration In passato gli sviluppatori aspettavano lunghi periodi di tempo prima di integrare le proprie modifiche con quelle degli altri membri del team. Di conseguenza, il lavoro di integrazione risultava spesso faticoso, soggetto all'errore.

Con la *Continuous Integration* (CI) i membri di un team aggiungono regolarmente modifiche ad un repository centralizzato (usando un sistema di version control, ad esempio *git*) [1]. Ad ogni nuovo commit, le nuove modifiche vengono intercettate da un server specializzato (CI server), dove un tool automatico esegue una build e una serie di *unit test* per verificare che la nuova modifica non contenga errori. Se si riscontrano problemi (il codice non compila o non supera i test) il commit viene rifiutato e lo sviluppatore riceve indicazioni su quali errori sono stati rilevati. In caso contrario la build può continuare il percorso all'interno della pipeline.

La CI costituisce il primo stadio della deployment pipeline ed è di fondamentale importanza per ridurre i tempi di convalida e per risolvere tempestivamente i bug.

Continuous Delivery Una volta superata la fase iniziale di integrazione e di test, la build completa viene fatta passare attraverso un certo numero di ambienti per essere testata ulteriormente. Obiettivo della *Continuous Delivery* (CD) è decidere, con un alto grado di confidenza, se la versione corrente sia pronta per essere messa in produzione o meno. In questo modo si avrà sempre a disposizione una build pronta per il deployment finale.

Ogni ambiente è progressivamente più simile all'ambiente di produzione e prevede test sempre più restrittivi, ma anche più lenti. Non si eseguono più semplici unit test, ma test dell'interfaccia, test delle performance, test di integrazione, test di regressione e così via. La scelta di eseguire le tipologie di test in ordine crescente di tempo permette di rendere più immediato il processo di bug-fixing. In questo modo se si riscontrano bug per un certo stadio della pipeline, non sarà necessario eseguire i test per gli stadi successivi, accorciando ulteriormente i tempi. È importante sottolineare che, in alcuni casi e per rendere l'intero processo più rapido, alcuni stadi possono essere eseguiti parallelamente, mantenendo tuttavia l'isolamento tra gli ambienti.

Se la build supera tutti gli stadi della pipeline, viene considerata valida per il deployment. A questo punto l'organizzazione ha due possibilità: potrebbe decidere di dispiegare automaticamente ogni versione che supera la deployment pipeline (in tal caso si parla di *Continuous Deployment*) oppure scegliere manualmente. In entrambi i casi la continuous delivery rende tale decisione una semplice scelta di business.

1.3.4 Uso di strumenti per l'automazione

Il processo di sviluppo del software ha a che fare quotidianamente con azioni delicate come l'integrazione del codice o il deployment dell'applicazione. Azioni di questo tipo erano storicamente eseguite da personale esperto, specializzato quindi nel portarle a termine correttamente secondo procedure standard. Al giorno d'oggi però, con la crescente complessità delle soluzioni richieste dal mercato, eseguire tali operazioni manualmente può portare facilmente a inconsistenze dovute in gran parte all'enorme quantità di variabili da tenere in considerazione. Avendo come scopo l'agilità nel consegnare soluzioni funzionanti ai clienti, non è più tollerabile la presenza di questo genere di inconsistenze.

La soluzione messa in atto della cultura DevOps, come già accennato nelle sezioni precedenti, è quella di *automatizzare* tutte le procedure che sono maggiormente soggette all'errore umano. In ambito DevOps con il termine automatizzare si intende, principalmente, rendere una qualsiasi procedura eseguibile tramite il lancio di un *singolo comando definito in maniera standard a livello di progetto*. La presenza di un solo comando rende improbabile l'errore umano e limita fortemente la possibilità di creare le incongruenze descritte in precedenza. In aggiunta, la *standardizzazione* delle procedure rispetto al progetto assicura che ogni volta che sia necessaria l'esecuzione di una procedura, questa venga sempre eseguita secondo gli stessi criteri, in maniera ripetibile. Solitamente questo comportamento si ottiene creando degli *script* trattati come parte integrante del progetto, ovvero tracciati dal controllo di versione e rispettanti gli stessi criteri di qualità del codice stesso, oppure tramite strumenti di terze parti.

Di seguito si analizzano le principali procedure soggette ad automazione.

Integrazione Quando è necessario integrare i commit di uno sviluppatore nel progetto intero, si fa uso di strumenti di version control come *git* in unione a servizi che mettono a disposizione repository remoti (*GitHub*, *BitBucket* o *GitLab*). Infine, per la continuous integration, prodotti come *Jenkins*, *Travis CI* o *Bamboo* si affiancano a quelli già citati.

Build Anche il processo di build viene affidato a tool automatici. Un esempio è costituito dai popolari *Gradle* e *Maven*. I tool per la build automation consentono di eseguire il cosiddetto *packaging* delle applicazioni, ovvero la creazione di un singolo artefatto pronto per l'esecuzione. Durante questo processo occorre reperire eventuali dipendenze dell'applicazione e compilarle a loro volta, creando un struttura di dipendenze ad albero. Scopo di questi tool è risolvere l'albero delle dipendenze in modo automatico e produrre l'artefatto eseguibile finale.

Testing Come discusso nella Sezione 1.3.3, anche parte del testing viene eseguito tramite script. Per questa fase si usano utilities come *JUnit* o *Selenium*. L'automazione di questa fase consente uno sviluppo molto più agevole e una scoperta degli errori molto più affidabile.

Deployment La fase di deployment presenta alcune peculiarità che la rendono particolarmente delicata.

Innanzitutto l'ambiente di destinazione potrebbe non contenere delle dipendenze esterne richieste dall'applicazione. Per risolvere questo problema in genere le applicazioni vengono ulteriormente "impacchettate" all'interno di *macchine virtuali* o, più recentemente, di *software container* (descritti in dettaglio nel Capitolo 3). Entrambe le tecnologie consentono di "preparare" degli artefatti detti *immagini* che contengono tutte le informazioni necessarie e sufficienti all'esecuzione dell'applicativo, incluso l'applicativo stesso. A partire da un'immagine è possibile creare un'insieme di istanze che vengono messe in esecuzione, a questo punto con tutte le dipendenze necessarie.

L'altra problematica tipica della fase di deployment è la gestione della cosiddetta *business continuity*. In molti casi non ci si può permettere che il servizio offerto venga disattivato, a volte neanche per brevissimi periodi di tempo, mentre generalmente il processo di deployment potrebbe richiedere diverse ore. Esistono varie tecniche per evitare questa condizione, ma vengono in genere affidate a tool automatici.

1.3.5 Monitoraggio pervasivo

DevOps prevede che sia il software, sia l'infrastruttura su cui esegue vengano costantemente monitorati. In fase di progettazione del sistema, il concetto di monitoraggio gioca un ruolo di grosso rilievo, perché impatta fortemente su tutta l'architettura. Innanzitutto, il sistema deve essere in grado di generare *log* dettagliati per qualsiasi tipo di attività o eventi che si verificano. In secondo luogo è necessario raccogliere i dati riguardo l'utilizzo delle risorse (hardware o

software che siano) e collezionarle in un *sistema di monitoraggio centralizzato*, in modo che vengano elaborate per la visualizzazione degli addetti.

Considerando che solitamente l' architettura complessiva del sistema è *distribuita*, scegliere una soluzione adeguata per il monitoraggio non è affatto banale. In un contesto a microservizi, ci si trova a dover gestire grandi flussi di dati provenienti da decine o addirittura centinaia di nodi, rendendo in questo modo proibitivo collezionare le informazioni in un repository centralizzato. Inoltre, un'altra *challenge* è costituita dalla frequenza con cui il software viene aggiornato (per via del continuous deployment), che rende difficile stabilire quali siano le condizioni di lavoro "normale".

Le tecniche con cui vengono raccolti i dati si possono dividere in tre categorie:

- **Agent-based**: ogni nodo contiene un *agente* (interno al sistema, ma esterno all'applicazione) che comunica periodicamente i dati di utilizzo al servizio centralizzato.

- **Agentless**: si fa uso di *protocolli specializzati* (es. SNMP) per comunicare i dati. In questo caso l'applicazione deve contribuire attivamente alla comunicazione.

- **Health checks**: in questo caso è un *servizio esterno* che interroga periodicamente il nodo attraverso i cosiddetti *health checks* (verifica dello stato di salute del sistema).

Un sistema necessita di essere monitorato per i seguenti motivi [2]:

- Identificare malfunzionamenti e/o crash dell'infrastruttura (fisica o virtualizzata) oppure dell'applicativo. In tal caso si potrà reagire di conseguenza per ristabilire il corretto funzionamento.

- Rilevare degradazione delle performance, misurando parametri come *latenza*, *throughput* e *utilizzo* e comparandoli con i valori *storici*.

- Prevedere le necessità future e pianificare di conseguenza l'allocazione delle risorse computazionali. Ciò rende possibile pianificare necessità a breve termine (per esempio il numero di repliche di un servizio) o a lungo termine (come l'allocazione dell'hardware).

- Determinare il grado di soddisfazione dell'utente a fronte di una modifica o in base all'affidabilità del sistema.

- Rilevare intrusione di utenti non autorizzati o attacchi informatici.

Capitolo 2

Architettura a microservizi

L'*architettura a microservizi* rappresenta uno stile architetturale emergente, grazie al quale ciascuna delle *sotto-parti atomiche* di un sistema viene isolata in un *applicativo indipendente*, che collabora con gli altri per creare il comportamento del sistema nel suo complesso. La collaborazione è data dalla possibilità che, per portare a termine una richiesta, un servizio possa a sua volta inviare ulteriori richieste ad altri servizi. A seconda dei casi quindi, ogni microservizio può essere visto come *consumatore* o *fornitore*. L'architettura può essere vista come un'evoluzione della *Service Oriented Architecture* (SOA), della quale preserva la natura distribuita, rendendola però più decentralizzata e partizionata nella gestione della logica applicativa. I microservizi nascono per via della crescente necessità, in accordo con gli scopi di DevOps, di avere un'architettura software che rispecchi maggiormente l'organizzazione dei team di sviluppo e che rifletta i bisogni di uno sviluppo più agile.

2.1 Evoluzione dalle applicazioni monolitiche

L'approccio tradizionale con cui le aziende producono le soluzioni informatiche è costituito da un'unica applicazione che racchiude al suo interno tutte le funzionalità necessarie, il cosiddetto *applicativo monolitico*. Ciascuna delle sue sotto-parti corrisponde ad un *modulo*, che implementa tutte le funzionalità di business per il dominio a cui appartiene. Inoltre, l'intera applicazione comunica per l'accesso ai dati con *un solo database* ed espone delle API con cui i client possono effettuare richieste. In Figura 2.1 è mostrata una tipica architettura di questo genere.

Questo è stato il tipico approccio alla programmazione per diversi anni in quanto, almeno inizialmente, costituiva un'architettura facile da sviluppare, testare e dispiegare. In realtà, ci si è presto resi conto che, a mano a mano che le dimensioni del software crescevano, quei processi che sembravano semplici

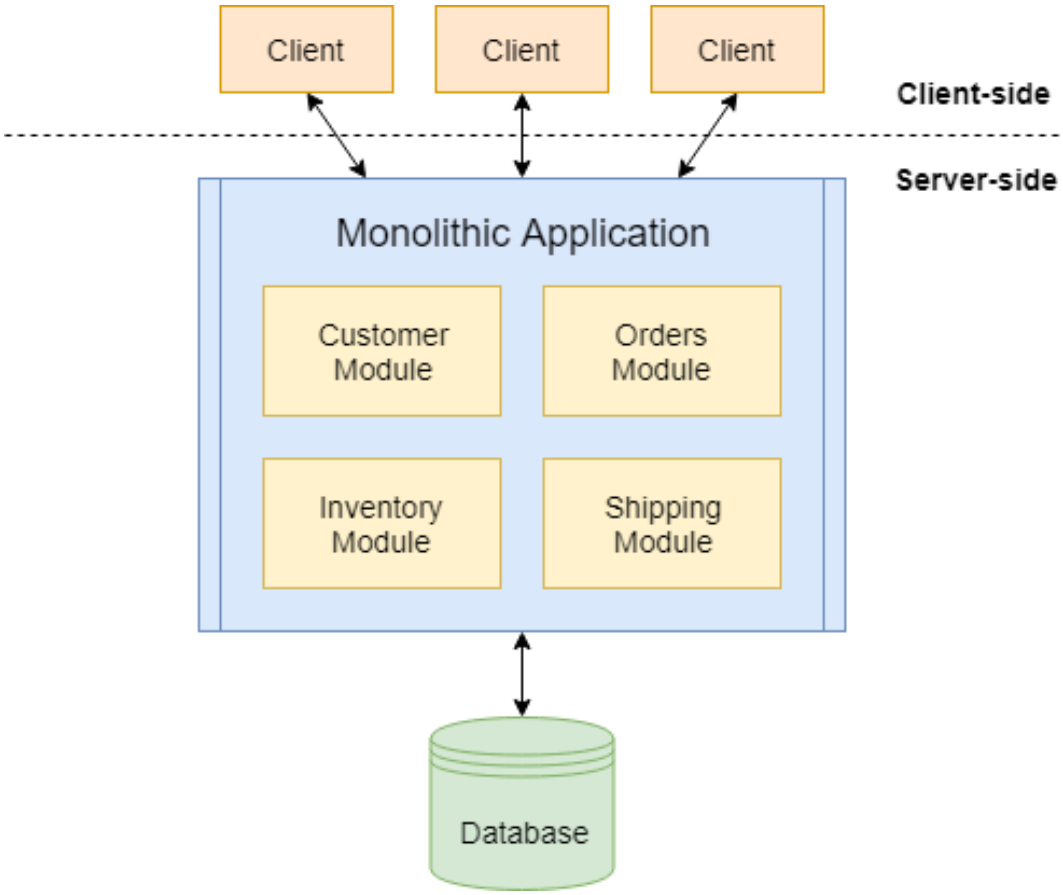


Figura 2.1: Schema architetturale di un applicativo monolitico

diventavano esponenzialmente più onerosi. Questo perché avere grandi team che lavorano alla stessa code base, integrare modifiche portate avanti per mesi, verificare che tale integrazione fosse priva di errori e infine dispiegare sui server grandi quantità di modifiche, risultava sempre più complicato. Veniva quindi a mancare il concetto di agilità e di indipendenza dei team. Chris Richardson chiama questa condizione il *monolithic hell* [9], per enfatizzare l'impraticabilità di gestire la crescita di un'applicazione monolitica.

Scegliere un approccio orientato ai microservizi risponde alla volontà di evadere dall'inferno monolitico. Ad alto livello, ciò consiste nel far diventare i moduli che costituivano il nucleo delle funzionalità processi a se stanti, che prendono il nome di microservizi (o più semplicemente servizi). Come si vedrà nella Sezione 2.2.2, ogni servizio avrà la propria base di dati a cui fare riferimento, separata da quelle degli altri servizi. Con lo scopo di unificare il punto di accesso ai servizi veri e propri, alcune architetture optano per l'inserimento di un *API Gateway*. Il suo scopo è quello di rendere i frontend del sistema più indipendenti possibili dalla struttura interna del backend. È importante sottolineare che, ai fini di mantenere la decentralizzazione del sistema, il gateway non aggiunge in nessun modo funzionalità di business e viene visto dai servizi reali come un semplice client.

Lo schema mostrato in Figura 2.2 mostra una possibile trasformazione dell'architettura precedente.

2.2 Derivazione dei principi da DevOps

Dovendo le sue origini in gran parte al movimento DevOps, non è una sorpresa che i principi che stanno alla base dell'architettura a microservizi siano mirati ad attuare le sue pratiche. Come anticipato nella Sezione 1.3.2, i microservizi nascono dall'esigenza dei team DevOps di ridurre al minimo la coordinazione tra team e di favorire la decentralizzazione del potere decisionale, mantenendo allo stesso tempo la confidenza che prodotti sviluppati da ciascuno cooperino correttamente.

Prima di sviscerare i dettagli e le motivazioni di ogni principio è utile avere una visione d'insieme di quali essi siano:

- **Decomposizione atomica del dominio**: ogni servizio è responsabile di un sottoinsieme del dominio non ulteriormente divisibile semanticamente ed espone tutte le funzionalità secondo un'interfaccia ben definita.

- **Incapsulamento**: la logica con cui un servizio è implementato deve essere nascosta agli altri servizi, che possono accedere a tali funzionalità esclusivamente tramite l'interfaccia esposta dal servizio stesso.

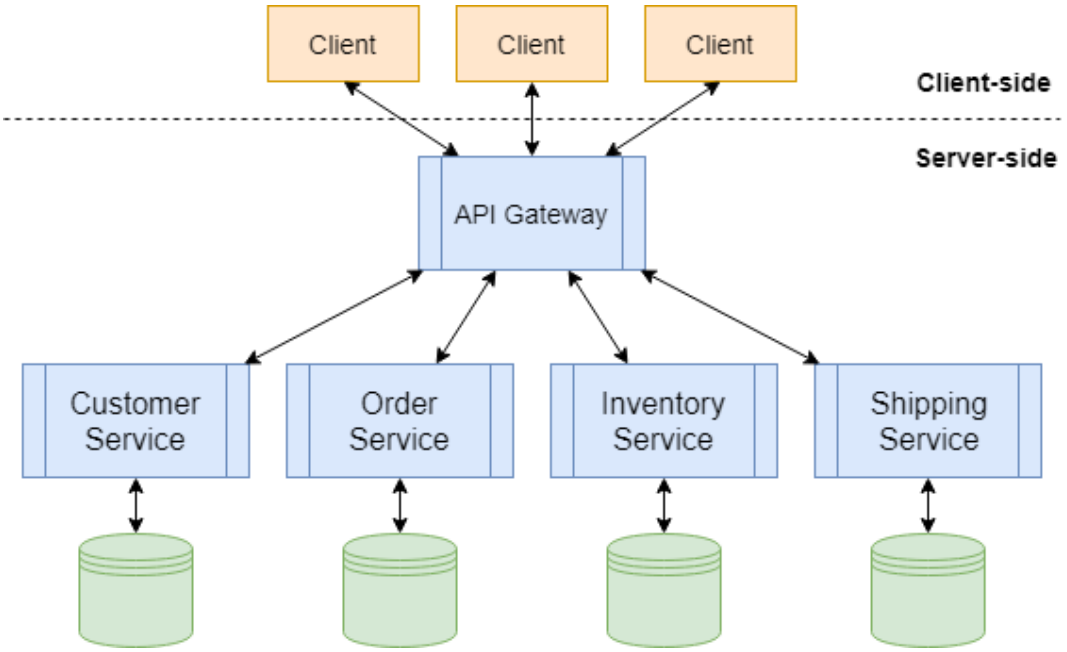


Figura 2.2: Evoluzione dell'architettura monolitica in un ecosistema di microservizi con API Gateway

- **Organizzazione e indipendenza dei team**: l'organizzazione aziendale in team rispecchia l'architettura del sistema, perciò i team sono di dimensione ridotta e la responsabilità di un servizio è affidata a un solo team. I servizi vengono implementati usando le tecnologie scelte dal team responsabile e, idealmente, le modifiche su un servizio non impattano sugli altri.
- **Decentralizzazione delle responsabilità**: sia i team, sia i servizi devono essere resi più responsabili di aspetti trasversali, piuttosto che fare affidamento a framework centralizzati.
- **Automazione dei processi e dell'infrastruttura**: affidare processi delicati come il testing e il deployment a tool automatici.

2.2.1 Decomposizione atomica del dominio

Con la SOA tradizionale si era portati a credere che, dovendo decidere il modo in cui suddividere il sistema in parti da assegnare ai vari team, la soluzione migliore fosse la divisione orizzontale in livelli di tecnologie (i livelli di *presentation*, *business logic* e *data access*). Ciò portava a due principali conseguenze. In primo luogo, ciascun team era composto esclusivamente da

esperti di una sola tecnologia. In secondo luogo, ogni nuova modifica, anche di piccole dimensioni, poteva generalmente toccare più layer dell'architettura e richiedere dunque la collaborazione di più team.

I microservizi si pongono l'obiettivo di partizionare l'applicazione verticalmente, assegnando ad ogni servizio un particolare dominio di business. I team acquisiscono skill provenienti da più aree aziendali, sono a stretto contatto con un solo contesto ben definito e coeso ed infine necessitano meno coordinazione a fronte di piccole modifiche che ricoprono solo un dominio di business.

I benefici di questo tipo di suddivisione si osservano anche a livello di modellazione e progettazione. L'alta coesione tra le entità e le funzionalità racchiuse in uno stesso contesto tende a rendere le interfacce dei servizi più stabili, richiedendo più raramente aggiornamenti dei servizi che dipendono da tali interfacce. Anche la modularità del sistema è ampiamente valorizzata. Un sistema in cui l'unità di decomposizione sono i sotto-domini riesce a sviluppare nuove capacità semplicemente componendo i servizi in maniera diversa. Qualora le specifiche richiedessero l'aggiunta di nuovi servizi, cosa non affatto rara nei software moderni, sarebbe molto più agevole inserirli nel sistema.

Naturalmente con tutto un insieme di benefici, il partizionamento verticale del sistema comporta una serie di problematiche non trascurabili. Scegliere adeguatamente i confini della suddivisione non è affatto semplice. Una buona progettazione dei confini deve essere tale da rendere i domini il più possibile disaccoppiati, ma allo stesso tempo mantenere un buon livello di modularità. Fallire nel tracciamento dei confini può sfociare in un'architettura globale esageratamente aggrovigliata, poiché gli aggiornamenti possono ricoprire frequentemente più di un servizio, annullando così tutti i benefici offerti dai microservizi in termini di indipendenza.

2.2.2 Incapsulamento

Il concetto di incapsulamento in ambito microservizi richiama fortemente quello di cui si sente parlare nel *paradigma a oggetti*. Nel contesto di quest'ultimo, ogni oggetto espone un'interfaccia pubblica contenente un certo numero di metodi, che costituiscono l'unico modo di accedere ai dati racchiusi nell'oggetto stesso.

Lo stesso concetto può essere esteso ai microservizi. In entrambi i casi, infatti, l'obiettivo è quello di nascondere l'organizzazione interna dei dati, rendendola accessibile tramite un insieme finito di operazioni prestabilite e promuovendo il *loose coupling*. Nel mondo dei microservice, le chiamate a metodo sono sostituite da richieste che viaggiano sulla rete e che utilizzano il protocollo stabilito dall'interfaccia del servizio. Ogni servizio deve quindi nascondere la porzione dei dati per il suo dominio di competenza.

Per ottenere tale comportamento la soluzione preferibile è avere un *database per servizio*. Le motivazioni per questa scelta implementativa sono molto semplici.

Si supponga di avere una certa quantità di dati appartenente a una specifica area del dominio, memorizzata in un database a cui è associato un certo microservizio. Il database potrebbe utilizzare una tecnologia qualsiasi. Si supponga inoltre che ci sia un certo numero di microservizi che, per portare a termine le proprie funzionalità, necessitano di interrogare la base di dati. Un primo approccio, incorretto, sarebbe quello di permettere a tutti i microservizi interessati di accedere al database, violando quindi il principio di incapsulamento. Ci sono diversi motivi per evitare questa soluzione, tra cui il principale è la possibile necessità di effettuare cambiamenti al database. Per esempio, se fosse richiesto di cambiare la tecnologia alla base della persistenza dei dati per motivi di costi o per avere un miglioramento delle performance, oppure se il formato di salvataggio dovesse essere cambiato, tutti i servizi dipendenti dovrebbero essere aggiornati per far fronte a tale modifica.

Avendo, invece, i dati accessibili solo contattando il servizio tramite l'interfaccia pubblica, gli sviluppatori hanno la possibilità di effettuare modifiche semplicemente aggiornando l'implementazione del servizio, ma mantenendone inalterata l'interfaccia pubblica e senza richiedere ulteriori cambiamenti per gli altri servizi.

Altre soluzioni percorribili oltre alla corrispondenza one-to-one tra service e database esistono, ma sono fortemente sconsigliate. Per esempio, considerando un database relazionale, si potrebbe pensare di creare tutte le tabelle dell'intero sistema e fare in modo che i servizi utilizzino esclusivamente le tabelle di propria competenza. In questo modo si è resilienti a piccole modifiche dello schema, ma non ad un'intera sostituzione del DBMS. In più il server dedicato a mantenere la persistenza dei dati sarebbe un grosso collo di bottiglia per ogni richiesta proveniente da tutti i servizi, degradando quindi le performance del sistema globale.

Il principio dell'incapsulamento, ai fini dell'agilità nello sviluppo, gioca un ruolo di vitale importanza. Esso permette agli sviluppatori di prendere decisioni in quasi totale autonomia. L'unico aspetto che richiede comunque coordinazione all'esterno del team riguarda esclusivamente la progettazione delle API pubbliche.

2.2.3 Organizzazione e indipendenza dei team

Riprendendo la linea di pensiero di DevOps, si può affermare che i microservizi siano nati per avvicinare il più possibile l'architettura del sistema da

sviluppare all'organizzazione aziendale, con lo scopo di trarne benefici in fatto di coordinazione e agilità decisionale.

Si è visto nella Sezione 1.3.2 che DevOps incoraggia la formazione di team di piccole dimensioni. Si è anche visto che è necessario che il carico di lavoro assegnato a ciascun team sia proporzionale a tali dimensioni, ma che il risultato finale abbia un alto grado di interoperabilità con il lavoro prodotto da altri team. L'architettura a microservizi rispetta fortemente queste necessità.

Partizionando finemente il dominio di un sistema, si vengono infatti a creare gruppi di funzionalità di dimensioni adeguate a essere sviluppate da team con pochi sviluppatori. Inoltre, se il principio di incapsulamento è adeguatamente rispettato, sarà sufficiente una sporadica, tuttavia mirata, coordinazione tra i team per ottenere componenti in grado di collaborare efficacemente. Ogni team sarà in grado di progettare le funzionalità del proprio servizio di competenza in maniera più agevole, perché dovrà avere conoscenza solamente di una piccola parte del dominio globale, rendendo l'intero processo più rapido.

2.2.4 Decentralizzazione

Quando si parla di decentralizzazione nel contesto dei microservizi, esistono tre diversi ambiti per tale termine: *decentralizzazione della logica*, *decentralizzazione dei dati* e *decentralizzazione dell'amministrazione*.

Decentralizzazione della logica Come già accennato nell'introduzione al capitolo, l'architettura a microservizi vuole, al contrario della classica SOA, rendere la logica applicativa più decentralizzata possibile. Questo perché lo sviluppo di sistemi SOA complessi tendeva generalmente a ridursi all'integrazione di applicazioni monolitiche, che utilizzavano infrastrutture software (ad esempio l'Enterprise Service Bus, ESB) che si facevano carico di gestire parte della già complicata logica. L'inevitabile conseguenza dell'affidare parte della logica di business a un bus centralizzato, fu quella di accoppiare strettamente i servizi e il sistema con il middleware di comunicazione, rendendo quindi il lavoro dei team sempre meno autonomo.

L'evoluzione dei microservice mira al concetto di *smart endpoints and dumb pipes* [7]. Viene soppiantato definitivamente l'uso di middleware come ESB, sostituendolo con tecnologie lightweight il cui unico scopo è il trasferimento di messaggi tra endpoint. Tali tecnologie sono totalmente inconsapevoli del contenuto informativo dei messaggi scambiati (*dumb pipes*), del quale sono invece a conoscenza solo i due estremi interessati nella comunicazione (*smart endpoints*).

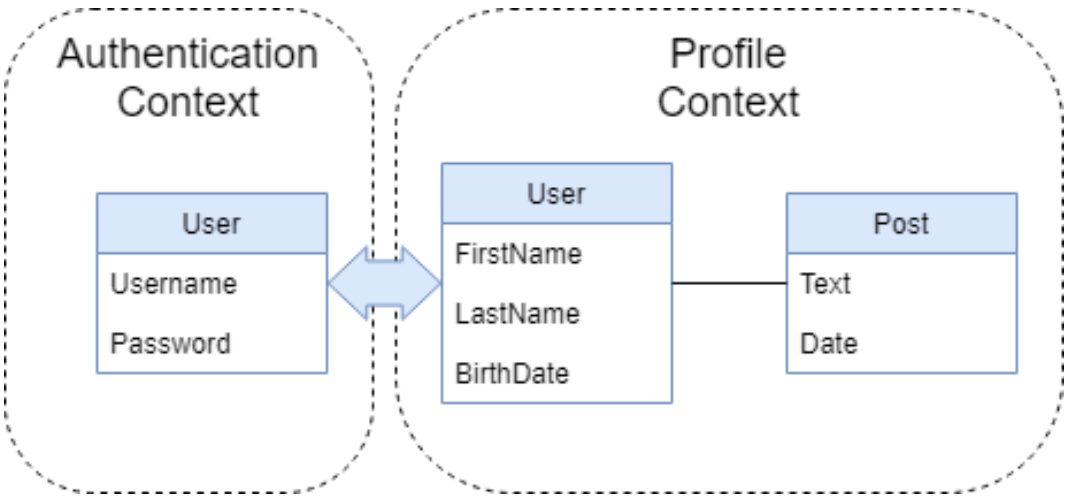


Figura 2.3: Esempio di entità condivisa tra contesti

Decentralizzazione dei dati Si è discusso nella Sezione 2.2.2 di come, ai fini dell'incapsulamento, ogni servizio gestisca autonomamente il proprio database nascondendolo al resto del sistema. Questo genera una forte decentralizzazione nella gestione dei dati, che si riflette su alcuni aspetti sia di implementazione, sia di modellazione.

Le conseguenze in fatto di implementazione consistono nella necessaria gestione della consistenza dei dati (descritta più in dettaglio nella Sezione 2.5.2) a fronte di operazioni di scrittura che riguardano più di un servizio. I servizi dovranno quindi collaborare tra loro per ottenere operazioni che non lasciano il sistema in stato inconsistente.

Diversa è invece la conseguenza della decentralizzazione dei dati sulla modellazione degli stessi. La suddivisione atomica del dominio prevede che il mondo che si vuole modellare venga partizionato in aree atomiche e semanticamente coese. Ciascuna di tali aree avrà quindi un *modello concettuale* del sistema globale diverso dagli altri. Alcune delle entità appartenenti al sistema apparterranno a uno solo dei sotto-domini individuati. Altre potrebbero essere invece condivise tra più contesti, avendo tuttavia semantiche leggermente differenti (pur trattandosi della stessa entità). Per esemplificare, si può pensare al significato che l'entità *Utente* può assumere in diversi contesti di un social network. Si supponga che la suddivisione del dominio produca i contesti di *Autenticazione* e *Profilo* (Figura 2.3). Nel primo caso l'entità *Utente* avrà la semantica di possessore di credenziali. Nel secondo caso invece l'entità *Utente* sarà costituita dai suoi dati anagrafici e i suoi post.

La possibilità che ci siano diverse visioni della stessa entità porta al bisogno di un metodo efficace per formalizzare i confini tra sotto-domini e le varie

semantiche attribuite alle entità. Si introduce il concetto di *Bounded Context*, proveniente dall'ambito del *Domain-Driven Design* (DDD), con l'obiettivo finale di avere un'associazione *one-to-one* tra bounded context e servizio. Il DDD suddivide modelli di dimensioni troppo elevate in aree dette Bounded Context, definendo esplicitamente le relazioni che intercorrono tra di essi [3]. Inoltre, per ciascun bounded context viene definito un insieme di regole e di terminologie che devono essere applicate ad esso. In questo modo, la terminologia e la semantica associate ad un'entità possono essere meglio comprese nella coordinazione interna e esterna dei team.

Decentralizzazione dell'amministrazione Rendere i team più responsabili di aspetti amministrativi del progetto è sicuramente uno dei principi chiave del movimento DevOps, specialmente in ottica microservizi. Applicazioni di grandi dimensioni rendono spesso indispensabile la possibilità di scegliere la tecnologia più adeguata a risolvere un determinato problema. Un approccio centralizzato all'amministrazione tende a rendere una certa tecnologia e/o framework lo standard da utilizzare a livello di progetto e porta gli sviluppatori a rallentare nella programmazione.

Scegliendo di rendere ogni team responsabile della tecnologia ritenuta migliore per portare a termine un lavoro, contribuisce a incrementare l'efficienza.

2.2.5 Automazione dei processi e dell'infrastruttura

In accordo col movimento DevOps, i microservizi abbracciano la cultura dell'automazione dell'infrastruttura. Se gestire manualmente processi come il deployment, il monitoraggio o il testing è complicato per una singola applicazione, gestirli per decine di applicazioni tra loro risulta impraticabile. L'obiettivo che si vuole raggiungere in un'ottica di microservizi è quello di avere la certezza che dopo la messa in produzione il sistema funzioni come previsto, sia considerando le singole parti, sia considerando l'interazione tra esse.

L'ambiente dei microservizi, in costante evoluzione in termini sia di funzionalità, sia di scala, rende le pratiche di automazione introdotte da DevOps (descritte nella Sezione 1.3.4) uno strumento fondamentale. Data la moltitudine dei componenti da installare, diventa indispensabile fare affidamento alla CI/CD, togliendo dal personale il fardello di gestire deployment altamente frammentato.

Un altro processo largamente automatizzato è il monitoraggio del sistema. In particolar modo si cerca di supervisionare lo stato di funzionamento delle macchine su cui eseguono i vari servizi, in modo che si possano sostituire le macchine che smettono di funzionare senza l'intervento umano.

2.3 Deployment indipendente

Tra i vantaggi che i microservizi offrono rispetto alle applicazioni monolitiche, il più importante è forse quello del *deployment indipendente* dei singoli servizi. L'indipendenza si ottiene in due ambiti: *indipendenza negli aggiornamenti* e *indipendenza nella scalabilità*.

2.3.1 Indipendenza negli aggiornamenti

Sam Newman afferma che *dovrebbe essere la norma, non l'eccezione, che un cambiamento effettuato su un servizio venga messo in produzione senza dover cambiare gli altri servizi* [7]. È questa la motivazione che porta al principio dell'incapsulamento. Tuttavia, il disaccoppiamento delle singole componenti rappresenta una condizione necessaria, ma non sufficiente ad avere tale capacità.

L'eventualità che le interfacce pubbliche debbano essere modificate, con la naturale conseguenza di diventare incompatibili con tutti i consumatori dipendenti da esse, costituisce un ostacolo verso l'indipendenza negli aggiornamenti. Tale situazione renderebbe la creazione di nuove versioni un processo oneroso, che coinvolge troppi servizi e che viola il principio di indipendenza. Una soluzione è data dal concetto di *Consumer-Driven Contract* (CDC). La tecnica consiste nella creazione, da parte dei consumatori di un servizio, di una serie di test automatici che verificano che il contratto atteso sia rispettato. Se una modifica dovesse far fallire un CDC, verrebbe immediatamente revocata, poiché infrangerebbe le aspettative del consumatore. È buona prassi che tali test vengano sviluppati addirittura prima che il servizio stesso venga implementato, in modo che i responsabili si pongano l'obiettivo di conformarsi al contratto imposto dai consumatori. L'uso dei CDC viene quindi integrato nella deployment pipeline ed assicura che le interfacce tra consumatori e fornitori siano sempre rispettate.

Tuttavia violare un contratto potrebbe essere inevitabile, perché richiesto ai fini dell'evoluzione funzionale del servizio. In un caso simile, l'unica alternativa per riuscire comunque a effettuare il deployment delle nuove funzionalità senza alterare i consumatori è mantenere attive più versioni dello stesso servizio e lasciare ai consumatori il tempo di far fronte agli aggiornamenti.

2.3.2 Indipendenza nella scalabilità

Uno svantaggio rilevante nei software monolitici è dato dal limitato controllo sulla scalabilità. Generalmente, i servizi basati sul paradigma *client-server* vengono scalati semplicemente replicando il processo server e nascondendo il

cluster così creato dietro un *load balancer*, ovvero un middleware che si occupa di bilanciare le richieste tra i vari processi server. Per un software monolitico ciò significa che ogni funzionalità è replicata lo stesso numero di volte. In realtà questo può costituire uno spreco di risorse: non tutte le funzionalità dell'applicazione vengono utilizzate necessariamente con la stessa frequenza.

Ancora una volta l'adozione dei microservizi può fornire una soluzione a questo fenomeno. Avendo ogni gruppo di funzionalità separata in un processo diverso, è semplice replicare ogni servizio indipendentemente e in base alle necessità di business.

Percorrendo questa strada tuttavia, il numero di servizi che vengono installati su un singolo host può generare effetti indesiderati dovuti alla condivisione di risorse, che siano hardware o software. Per ridurre al minimo i *side effects*, la soluzione preferibile sarebbe quella di avere una singola istanza di servizio per host. Tuttavia, in qualche caso ciò potrebbe produrre un costo troppo elevato, specie quando le macchine utilizzate sono fisiche. In questo frangente entrano in gioco la virtualizzazione e i servizi di hosting cloud, che rendono meno proibitivi i costi di provisioning dell'infrastruttura.

2.4 Tecnologie di comunicazione

Si è fatto riferimento più volte al fatto che i processi di un ecosistema a microservizi debbano comunicare tra di loro con lo scopo di comporre le loro capacità. La tipologia di comunicazione e il relativo protocollo possono essere classificati lungo due assi [5]. Un sistema a microservizi usa tipicamente più di una tipologia di comunicazione, a seconda delle necessità.

Comunicazione sincrona VS comunicazione asincrona Un protocollo sincrono stabilisce che un client effettui una richiesta a un servizio e che si metta in attesa di una risposta. HTTP è un esempio di protocollo sincrono, che viene largamente utilizzato nei microservizi arricchito dallo stile RESTful. Un protocollo asincrono invece non prevede l'attesa di una risposta. Il client invia sempre dati al server, ma subito dopo prosegue nell'esecuzione dei suoi task. Ne è un esempio il protocollo AMQP. Si osservi che la distinzione tra comunicazione sincrona e asincrona non tiene conto dell'implementazione dell'attesa del cliente (thread bloccato o meno), ma semplicemente della natura del protocollo.

Comunicazione single-receiver VS comunicazione multi-receiver Si parla di single-receiver quando la comunicazione viene svolta con esattamente un interlocutore, il fornitore del servizio. Se, al contrario, un messaggio

può essere intercettato e gestito da un numero indefinito di entità si avrà comunicazione multi-receiver. Quest'ultima tipologia viene spesso utilizzata per meccanismi *publish/subscribe*, per la creazione di ecosistemi *event-driven*.

2.5 Principali problematiche

Con tutto un insieme di evidenti benefici, l'architettura a microservizi porta con sé anche alcune problematiche che gli sviluppatori devono risolvere. Esse sono dovute principalmente alla natura ampiamente distribuita dell'ambiente dei microservizi.

2.5.1 Gestione dei guasti e della latenza

Uno dei problemi che ogni sistema distribuito si trova ad affrontare è l'inaffidabilità della rete. Gli sviluppatori di questo tipo di sistemi hanno da sempre dovuto progettare soluzioni che tenessero conto della latenza e della sua variabilità e imprevedibilità nel tempo. Quando si fa uso di protocolli di comunicazione sincrona, ovvero in cui è necessario attendere una risposta a fronte di ogni richiesta, il problema della variabilità della latenza costituisce una grande causa di problemi. Ciò è dato dal fatto che, per un client che effettua una richiesta, non è possibile in alcun modo determinare se è il destinatario a essere malfunzionante o la rete a essere congestionata. Il problema della latenza costituisce una difficoltà ancora maggiore per un sistema a microservizi, dato che per risolvere una richiesta sono sovente necessarie ulteriori richieste interne. In tal modo, ogni ritardo si accumulerà, rendendo l'intero sistema sempre più congestionato.

La soluzione comunemente adottata è l'uso di *timeout* per le richieste, ovvero un periodo di tempo dopo il quale il mittente considera la richiesta fallita. Cruciale è la scelta della durata di tale periodo: un periodo troppo breve può provocare fallimenti delle richieste nei casi in cui il destinatario o la rete sono solamente lenti nel recapitare la risposta; un periodo troppo lungo provoca ritardi eccessivi nei casi in cui le richieste falliscono effettivamente.

Altre strategie da usare in aggiunta al meccanismo di timeout permettono, a fronte di malfunzionamenti frequenti della rete e dei servizi, di reagire in maniera più efficace. Due tra i pattern più comunemente usati sono il *Bulkheading* e il *Circuit Breaker*.

Bulkheading Tipicamente ogni servizio è dotato di un *thread pool* usato per gestire le richieste che invia ad altri fornitori. Se uno di essi fosse guasto per

qualche motivo, tutte le richieste si accumulerebbero in attesa di timeout e il thread pool verrebbe saturato velocemente, impedendo quindi alle altre richieste di essere gestite rapidamente. La tecnica di Bulkheading consiste nell'allocare un thread pool per ogni dipendenza esterna piuttosto che uno condiviso, in modo da isolare la congestione al solo destinatario malfunzionante.

Circuit Breaker Con questo stratagemma si fa in modo che, dopo un certo numero di timeout provocati da un servizio fornitore, ogni successiva richiesta verso di esso non sia mandata e sia direttamente considerata fallita. La conseguenza è che ogni servizio abbrevia i tempi nel gestire richieste che prima avrebbero richiesto di attendere l'intero periodo di timeout.

2.5.2 Consistenza dei dati

Una grossa carenza dovuta alla distribuzione dei dati si palesa quando un'operazione richiede di aggiornare dati appartenenti a più di un servizio. In particolar modo, sono da tenere in considerazione le operazioni che, ad un certo passo dell'esecuzione, potrebbero fallire a causa della violazione di un vincolo di business o per un malfunzionamento. Ciò che si dovrebbe fare in questi casi è annullare a ritroso le operazioni eseguite fino al momento del fallimento, per fare in modo di non avere dati inconsistenti. Tale processo è noto nel mondo dell'informatica con il nome di *transazione*. Per transazione si intende un insieme di operazioni eseguite su una base di dati che rispetta le proprietà del modello *ACID*: *Atomicity*, *Consistency*, *Isolation* e *Durability*.

- **Atomicity**: l'esecuzione della transazione non è divisibile e non può essere parziale.
- **Consistency**: lo stato dei dati è coerente sia prima dell'esecuzione della transazione, sia dopo.
- **Isolation**: ogni transazione non deve interferire con altre transazioni in esecuzione.
- **Durability**: alla fine della transazione tutti i dati sono salvati in maniera persistente.

Ogni transazione può terminare correttamente (*commit*) oppure essere annullata (*rollback*).

Generalmente i DBMS mettono a disposizione le funzionalità per poter eseguire sequenze di operazioni in maniera transazionale. In questo modo è possibile assicurare la consistenza dei dati all'interno del database stesso.

Quando però i dati sono distribuiti su più di un database non è più possibile fare affidamento a questi meccanismi.

La letteratura propone il concetto di *Saga* come alternativa alle transazioni per mantenere la consistenza. Una Saga è costituita da una sequenza di operazioni, ognuna delle deve essere eseguita da un solo servizio in maniera transazionale. A ciascuna operazione ne corrisponde un'altra che ne annulla gli effetti, detta *compensazione*. Il processo prevede che le singole operazioni vengano eseguite sequenzialmente e ciascuna di esse atomicamente. L'orchestrazione dei vari passaggi avviene tipicamente tramite pubblicazione di eventi; nello specifico, ogni volta che un servizio termina correttamente una transazione pubblica un evento che viene intercettato dal componente incaricato di far continuare la saga (che può essere un servizio centralizzato o il servizio responsabile della successiva operazione). Per raggiungere lo scopo finale è però indispensabile che la transazione e la pubblicazione dell'evento vengano eseguite in maniera atomica. Nel momento in cui una di esse fallisce, tutte le compensazioni delle operazioni precedenti vengono invocate in ordine inverso. Data la possibilità che il gestore delle saghe possa a sua volta subire malfunzionamenti, si usano meccanismi per far riprendere automaticamente il rollback una volta ristabilito il servizio.

Il modello di consistenza offerto da questo approccio non è più di tipo *ACID*, ma di tipo *BASE*: *Basic Availability*, *Soft state*, *Eventual consistency*.

- **Basic Availability**: in un determinato momento, i dati potrebbero essere in stato inconsistente. Il sistema in tali momenti deve comunque essere disponibile, pur rispondendo con messaggi di errore.

- **Soft state**: anche in periodi di assenza di input esplicito, lo stato dei dati potrebbe variare a causa dell'eventual consistency.

- **Eventual consistency**: è garantito che i dati diventeranno col tempo consistenti, anche se in un determinato momento potrebbero non esserlo.

Capitolo 3

Software Containers e Docker

La tecnologia della *containerizzazione* delle applicazioni sta prendendo piede sempre più velocemente nel mondo dell'informatica, tanto che molte aziende la considerano una prerogativa del loro processo di sviluppo. Con la diffusione del movimento DevOps infatti, la necessità di semplificare e standardizzare il processo di deployment con lo scopo di consegnare più velocemente nuove soluzioni ha fatto sì che questa nuova strategia diventasse indispensabile. Essa rappresenta un'alternativa leggera alla classica tecnologia della *virtualizzazione tramite macchine virtuali*, che mantiene tuttavia le caratteristiche fondamentali per cui le aziende si affidano a servizi di virtualizzazione. Questo capitolo vuole fornire una panoramica sulle motivazioni e sulle caratteristiche di questa recente innovazione, prendendo come riferimento la principale piattaforma in ambito di software container: *Docker*.

3.1 Virtualizzazione

Il termine *virtualizzazione* assume svariati significati nel contesto informatico, ma essi possono essere riassunti in un'unica definizione che ne fattorizza le caratteristiche. Per virtualizzazione si intende *l'emulazione tramite software di una qualsiasi risorsa che prende il posto dell'originale all'interno di un sistema*. L'obiettivo principale della virtualizzazione è quello di astrarre le risorse hardware in modo da ridurre i costi dovuti all'allocazione e al mantenimento dell'hardware stesso e contemporaneamente ottenere benefici in termini di flessibilità e scalabilità.

Sono oggetto di virtualizzazione reti, dispositivi di storage e memorie, ma l'ambito in cui se ne sente parlare più frequentemente è quello della virtualizzazione delle piattaforme per l'esecuzione di applicazioni, in particolare della virtualizzazione hardware-level e OS-level. Grazie a queste tecnologie è possibile effettuare un packaging standardizzato delle applicazioni, cioè creare

degli artefatti facilmente eseguibili in diversi ambienti e contenenti tutte le dipendenze necessarie all'applicazione stessa. Tali artefatti sono tipicamente chiamati immagini e, una volta messi in esecuzione, producono istanze di macchine virtuali nel caso hardware-level e software container nel caso OS-level. La capacità di effettuare in maniera standard il packaging delle applicazioni è fondamentale, poiché avere unità pronte per il deployment dà la certezza che il sistema non risenta dei trasferimenti tra un ambiente e un altro, processo frequente nella CI/CD.

Servizi di questo tipo vengono generalmente utilizzati nel dispiegamento di applicativi server-side, specialmente se distribuiti, per una serie di motivi. Innanzitutto, le istanze in esecuzione sono facilmente replicabili. Infatti per avviare una seconda copia di un'istanza in esecuzione è sufficiente riutilizzare la stessa immagine di partenza. Questo offre evidenti vantaggi nel momento in cui è necessaria un'efficiente scalabilità del sistema, anche in tempo reale.

Un altro aspetto rilevante è la portabilità delle immagini: essendo l'ambiente di esecuzione emulato via software, sarà sufficiente usare lo stesso tipo di emulatore per poter trasferire l'immagine tra sistemi diversi, con una buona confidenza che continui a funzionare correttamente.

Generalmente i servizi di virtualizzazione offrono anche la possibilità di effettuare le cosiddette *migrations*, ovvero il trasferimento di istanze in esecuzione tra un sistema fisico e un altro, a seguito di malfunzionamenti dell'hardware sottostante. Nell'ottica del monitoraggio e gestione dei guasti, avere questa possibilità rende istantaneo reagire ai guasti dell'hardware per ripristinare il funzionamento del servizio.

L'adozione di servizi di virtualizzazione nel dispiegamento di applicazioni server riduce fortemente i costi della fornitura dell'hardware. È sufficiente pensare a un sistema implementato a microservizi. Si è visto nella Sezione 2.3.2 come per fare in modo che i side effects siano minimi, sia preferibile che ogni processo esegua in un ambiente isolato dagli altri. Inoltre, il numero di servizi diversi è in genere nell'ordine delle decine e ciascun servizio potrebbe richiedere una quantità arbitraria di repliche. Sarebbe impensabile allocare una macchina fisica per ciascuno dei processi necessari. Grazie alla virtualizzazione è possibile creare più ambienti sufficientemente isolati e farli convivere all'interno della stessa macchina fisica. In questo modo le aziende ridurranno la quantità di hardware necessario e di conseguenza il costo di acquisto e mantenimento.

I costi di mantenimento dell'hardware possono essere definitivamente superati facendo affidamento a *servizi di hosting* sul cloud. Grazie ad essi, le aziende non devono più occuparsi del provisioning dell'hardware, ma possono richiedere che delle piattaforme di hosting si facciano carico di mettere in esecuzione le istanze virtuali secondo i parametri richiesti.

3.1.1 Hardware-Level virtualization

La virtualizzazione a livello hardware consente di emulare tramite software una macchina fisica, mostrando all'utilizzatore un'interfaccia su cui è possibile installare uno o più sistemi operativi, detti *sistemi operativi guest*.

La componente software che esegue l'emulazione è detta *hypervisor* e può essere installata direttamente sull'hardware (*Bare Metal Hypervisor*) oppure può essere ospitata dal sistema operativo già presente sulla macchina fisica (*Hosted Hypervisor*). Sull'hypervisor è possibile installare ed eseguire più sistemi operativi, ognuno dei quali condivide con gli altri le risorse hardware sottostanti. È compito dell'hypervisor garantire l'isolamento tra i sistemi e gestire l'accesso alle risorse condivise.

Un'istanza di macchina virtuale viene tipicamente avviata a partire da un file, detto *immagine della macchina virtuale*, che contiene il sistema operativo, le applicazioni installate e le loro librerie o dipendenze.

3.1.2 OS-Level virtualization: i Software Container

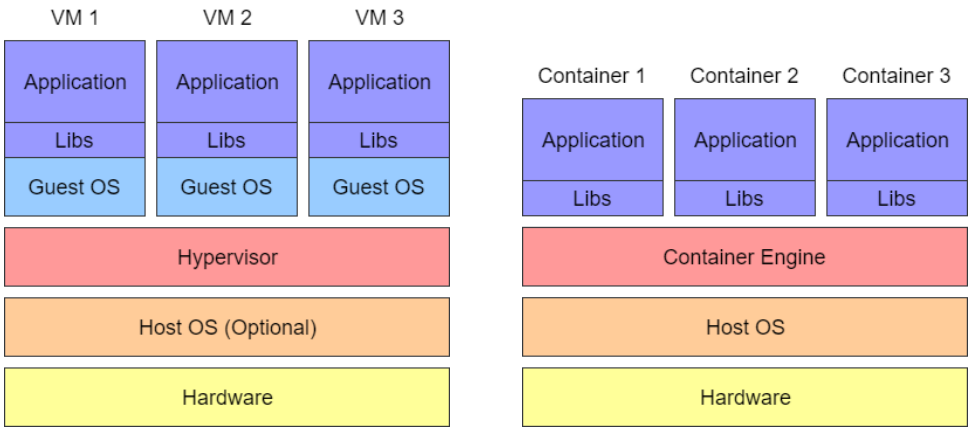
I software container si pongono un livello di astrazione sopra la macchina virtuale. La risorsa che viene emulata non è più la macchina fisica, ma il kernel del sistema operativo ospitante, perciò all'utente viene presentata una partizione isolata di tale sistema operativo sul quale è possibile installare applicazioni. Ciascuna partizione è detta *software container* o, più semplicemente, *container*. Come per le macchine virtuali, un container racchiude al suo interno le applicazioni e le loro dipendenze, ma questa volta non è più necessario avere un sistema operativo guest, poiché viene condiviso con quello del sistema host. Per questo motivo spesso di fa riferimento alla containerizzazione come a una versione *lightweight* della virtualizzazione tramite macchina virtuale.

Il compito di gestire l'esecuzione dei container è affidato ad un *Container Engine*, che può essere pensato come il corrispondente dell'hypervisor nella virtualizzazione OS-level.

Come le macchine virtuali, anche i container vengono tipicamente messi in esecuzione sulla base di file immagine che racchiudono una descrizione del contenuto del container.

3.1.3 Confronto tra macchine virtuali e container

Una differenza evidente tra le due tipologie di virtualizzazione si può notare osservando la Figura 3.1. L'uso delle macchine virtuali (Figura 3.1a) richiede che vi sia, oltre alle dipendenze dell'applicazione, un sistema operativo guest per ciascuna istanza, comportando un elevato overhead in termini di memoria. Al contrario, i container (Figura 3.1b) eseguono direttamente sul kernel del



(a) Macchine virtuali (b) Container

Figura 3.1: Confronto tra macchine virtuali e container

sistema ospitante e hanno quindi necessità di replicare esclusivamente le librerie specifiche per l'applicazione.

Altri vantaggi dei container rispetto alle macchine virtuali si hanno nell'ambito delle performance e dei tempi di avvio delle singole istanze. Un container non ha la necessità che ogni istruzione venga interpretata dall'hypervisor per essere eseguita sulla macchina fisica, poiché viene eseguita direttamente sul kernel host. In più, non è necessario che all'avvio venga caricato l'intero sistema operativo guest, assicurando un processo di provisioning pressoché istantaneo.

Le motivazioni sopra citate potrebbero far sembrare la virtualizzazione a livello OS preferibile sotto ogni aspetto. In realtà non sempre i container sono una soluzione percorribile.

Innanzitutto ogni applicazione deve necessariamente eseguire sullo stesso sistema operativo della macchina host. Se il sistema è composto da applicazioni di varia natura che richiedono di eseguire su sistemi diversi, i container non sono più adottabili. Inoltre, l'isolamento a livello di processo dato dai container non è allo stesso livello di quello totale delle macchine virtuali. Se un'applicazione all'interno di un container dovesse compromettere in qualsiasi modo il kernel, tutto il sistema cesserebbe di funzionare.

Naturalmente, essendo queste condizioni l'eccezione, non la norma, i container rimangono uno strumento di fondamentale importanza.

3.2 Docker

In questa sezione si vuole presentare la tecnologia che rappresenta lo standard de facto per la creazione e la gestione di software container su sistemi

Linux, la piattaforma *Docker*. Docker è un progetto *open source*, sviluppato dall'omonima azienda, *Docker Inc*, insieme al contribuito di una vasta *community*. L'enorme successo di Docker nelle grandi e piccole aziende è dovuto alla sua semplicità d'uso, insieme agli indiscussi vantaggi che comporta l'impiego della containerizzazione nei processi di sviluppo. Si vuole perciò fornire una panoramica dell'architettura e del funzionamento di Docker al fine di comprenderne più a fondo i benefici.

3.2.1 Architettura di Docker

L'architettura di Docker segue un paradigma client-server ed è composta ad alto livello da tre macro componenti: *Docker Engine/Daemon*, *Docker Client* e *Docker Registry* [4].

- **Docker Engine o Docker Daemon**: Rappresenta la componente server dell'architettura e gestisce le funzionalità centrali per gestire i container insieme ai *Docker objects*. Si tratta di un processo di tipo *long-running* (da qui il termine *daemon*) che espone delle REST API per permettere ai client di interagire con il ciclo di vita dei container e degli oggetti.

- **Docker Client**: Si tratta di un'interfaccia a linea di comando che, facendo affidamento alle REST API del Docker Daemon, permette agli utenti di interagire con la piattaforma. È importante sottolineare che il Client e l'Engine non devono necessariamente essere sulla stessa macchina host per comunicare.

- **Docker Registry**: Rappresenta un repository di immagini Docker salvate in remoto. Ogni client ha un Docker Registry di default da cui preleva o verso cui salva le immagini qualora vi fosse la necessità. *Docker Hub* è un repository pubblico messo a disposizione da Docker Inc.

3.2.2 Docker Objects

I *Docker Objects* sono le entità principali coinvolte nell'ecosistema Docker. Il ciclo di vita di ciascuno di essi viene gestito direttamente dal Docker Engine. Le entità usate più di frequente vengono elencate di seguito.

Docker Images Si tratta di template che vengono usati per creare e mettere in esecuzione i container. Tipicamente, le immagini vengono create a partire da file di testo chiamati *Dockerfile*, che contengono la sequenza di istruzioni usata per configurare l'immagine stessa. Ciascuna istruzione corrisponde ad

un layer dell'immagine e consente di specificare comandi da eseguire all'interno del container, copiare dati dall'host, modificare l'ambiente d'esecuzione, ecc. Inoltre, solitamente le immagini vengono costruite sulla base di altre immagini, che possono essere prelevate da un Docker Registry o essere già presenti in locale.

Docker Containers Un container rappresenta un'istanza in esecuzione di un'immagine ed è l'elemento base della virtualizzazione OS-level. Il Docker Engine espone delle API che permettono di creare, eseguire, fermare e addirittura spostare container con estrema semplicità. La configurazione di un container è principalmente contenuta nell'immagine da cui viene creato, ma alcuni parametri vengono specificati nel momento in cui viene lanciato. Un esempio di questi parametri è dato dalla Docker Network a cui il container deve essere agganciato, essendo un dettaglio sconosciuto a priori.

Docker Networks Sono le entità che consentono ai container di accedere alle funzionalità di rete, facendoli quindi di comunicare tra loro o con il mondo esterno. Ad ogni network è associato un *driver*, che ha lo scopo di decidere la modalità con cui i pacchetti vengono instradati al suo interno. A seconda del tipo di driver è possibile creare reti interne all'host (ad esempio il *bridge driver*) oppure reti che si estendono su più di un host (*overlay driver*). Ogni container può essere collegato a più reti, per ciascuna delle quali vedrà una diversa interfaccia e un diverso indirizzo IP.

Docker Volumes e Bind Mounts Ogni volta che un container viene eliminato, tutti i contenuti del suo file system vengono persi. In certi casi è però necessario che i dati siano persistenti anche a fronte della cancellazione. Docker lascia quindi la possibilità di mappare delle directory del file system del container a directory appartenenti all'host. Sia i Docker Volumes che i Bind Mounts offrono questo meccanismo. I volumi sono directory gestite dall'engine di Docker a cui viene associato un nome stabilito dall'utente, mentre con i bind mounts è possibile specificare directory arbitrarie del file system host sulle quali vengono mappate le directory del container. Il meccanismo dei volumi rappresenta una soluzione più flessibile in quanto, essendo gestita interamente da Docker tramite un nome simbolico, non richiede che il file system dell'host rispetti una struttura predefinita.

3.3 Container Orchestration

Di per sé, Docker consente di gestire agilmente piccoli gruppi di container che interagiscono tra loro in maniera semplice. I suoi comandi consentono di creare, eliminare o modificare i Docker objects singolarmente, ma nel momento in cui è necessario dispiegare un'architettura molto articolata e con un elevato numero di istanze, gestire manualmente ogni singolo aspetto del deployment risulta macchinoso, soggetto a errori. Sono quindi necessari degli strumenti, definiti *orchestratori*, che rendono il suo uso più flessibile e automatico, e che consentono di gestire i sistemi nella loro interezza.

Gli strumenti descritti nelle sezioni successive consentono di pianificare il deployment del sistema e successivamente di eseguirlo automaticamente, talvolta gestendo in tempo reale l'evoluzione e la scalabilità del sistema stesso. Ciò avviene dichiarando la struttura dei servizi in dei manifest, che vengono tracciati dal controllo di versione. La possibilità di raccogliere tutte le configurazioni dell'ambiente all'interno di file offre un vantaggio importante: diventa molto semplice riprodurre l'ambiente di esecuzione su infrastrutture diverse con la certezza che questo sia consistente con tutti gli altri, semplificando notevolmente le fasi della CI/CD. Gli orchestratori usati più frequentemente sono *Docker Compose*/*Docker Swarm*, sviluppati da Docker Inc stessa, e *Kubernetes*, sviluppato da Google.

3.3.1 Docker Compose

Docker Compose è uno strumento che permette di automatizzare il deployment di applicazioni multi-container su un singolo host, specificandone le configurazioni in dei manifest detti *compose files*, scritti in formato YAML. Grazie a questi ultimi si possono dichiarare i Docker Objects coinvolti nel dispiegamento del sistema e le relative proprietà.

Docker compose introduce l'astrazione di *Servizio*, un insieme di container che condividono la stessa immagine e la stessa configurazione. Ognuno di essi dispone di un nome univoco utilizzato per identificare il servizio stesso e renderlo facilmente accessibile agli altri container del sistema ed ha inoltre un load balancer che smista le richieste tra i vari container appartenenti al servizio. Questa entità viene spesso, ma non sempre, utilizzata per modellare il concetto di microservizio, in quanto si tratta di in un gruppo di repliche dello stesso componente software che vengono racchiuse da un'unica interfaccia.

Il compose file permette di configurare lo stack di servizi, indicando per ognuno parametri come la provenienza dell'immagine, le porte esposte, le variabili d'ambiente, le dipendenze da altri servizi, ecc.

Il Listato 3.1 mostra un esempio di compose file per un sistema contenente due servizi, di nome *redisdb* e *webapp*, collegati tramite una rete chiamata *mynet*.

```
version: '3.7'

services:
  redisdb:
    image: redis
    networks:
      - mynet
  webapp:
    build:
      context: ./mywebapp
    ports:
      - 8080:80
    networks:
      - mynet
    depends_on:
      - redisdb

networks:
  mynet:
```

Listato 3.1: Esempio di Compose File

La limitazione del deployment alla sola macchina locale rende Docker Compose uno strumento utilizzato principalmente dagli sviluppatori per testare rapidamente il sistema nella sua globalità. Qualora fosse necessario estendere il dispiegamento ad un cluster di computer sarebbe necessario utilizzare Docker Swarm.

3.3.2 Docker Swarm

Docker Swarm supera le limitazioni di Docker Compose, permettendo il dispiegamento di un sistema multi-container su un cluster di computer e la scalabilità dei singoli servizi su più host diversi. Swarm mantiene il formato e la sintassi del manifest utilizzato da Docker Compose, seppur con qualche piccola differenza dovuta alla diversa architettura.

Docker Swarm introduce inoltre i concetti di *nodo* e di *task*:

- **Nodo**: si tratta di un'istanza del Docker Engine che partecipa al cluster e che può avere il ruolo di *manager*, *worker* o entrambi.

- **Task**: rappresenta un'istanza di un servizio che deve essere messa in esecuzione su un nodo.

Quando è necessario dispiegare un servizio, il cui stato desiderato[1] è definito nel compose file, il master genera i task necessari e richiede ai worker di eseguirli e si aspetta che gli venga comunicato periodicamente lo stato di attività di tali task. È perciò compito del manager verificare il corretto funzionamento del sistema e, a fronte di guasti, fare in modo che lo stato desiderato venga ristabilito.

3.3.3 Kubernetes

Kubernetes, spesso abbreviato con *K8s*, può essere considerato il sistema di orchestrazione di container più utilizzato nel dispiegamento di applicazioni distribuite. Rispetto a Docker Swarm, offre scalabilità, fault tolerance e monitoraggio decisamente più robusti e mette a disposizione degli operatori un insieme di funzionalità più ampio. Una tra queste è la possibilità di scalare automaticamente un servizio in base al carico di lavoro misurato. Grazie a questa feature, Kubernetes diventa particolarmente adatto ad essere utilizzato in ambienti ospitati nel cloud, poiché evita lo spreco di risorse computazionali e di conseguenza riduce i costi al minimo indispensabile. Per contro, la sua architettura è sensibilmente più complessa e in alcuni casi può provocare un deployment leggermente più inefficiente rispetto a Swarm. Tuttavia, si tratta di un trade-off accettabile nella maggior parte dei casi.

L'architettura di Kubernetes (Figura 3.2) è composta da due tipi di entità. Il *Kubernetes Master* rappresenta il controllore del cluster e il suo compito è quello di mantenere lo stato del sistema più vicino possibile a quello desiderato. Il master espone delle API pubbliche che possono essere usate dagli operatori tramite strumenti come *Kubectl* per modificare lo stato del sistema o per ricevere informazioni su di esso. Sulla base dello stato desiderato il master decide in che modo schedulare l'esecuzione dei task. Ogni altro nodo che partecipa al cluster ha invece il ruolo di *Kubernetes Worker*. Essi comunicano periodicamente con il master per scambiare informazioni di controllo sul loro stato corrente e sui task da eseguire.

Per quanto riguarda l'architettura generale, Docker Swarm e Kubernetes sono estremamente simili. La disparità si trova principalmente nel modello del deployment adottato da quest'ultimo.

L'unità minima di deployment per Kubernetes è detta *pod* e consiste in un insieme di container che lavorano a stretto contatto tra loro e si vedono

[1]Per stato desiderato si intende la configurazione dei container del servizio e il numero di repliche richieste, che può in alcuni momenti differire da quello attuale perché il manager non ha ancora aggiornato i worker.

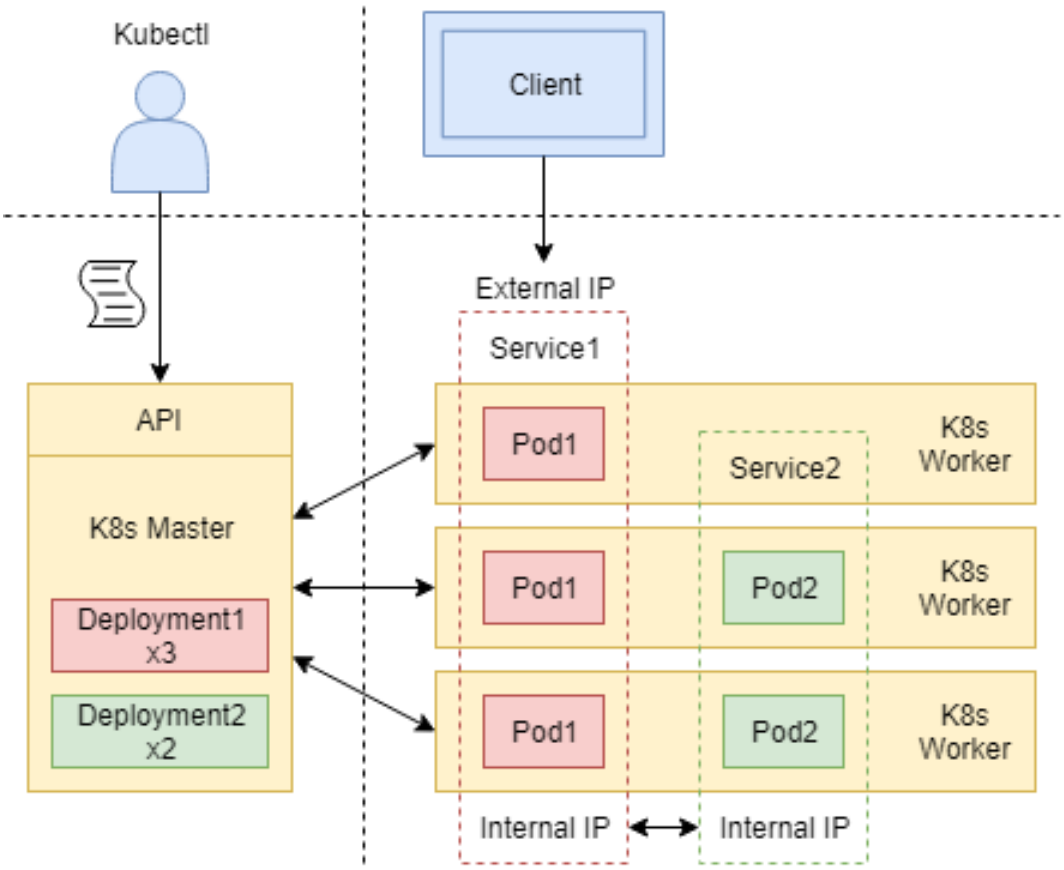


Figura 3.2: Architettura globale di Kubernetes

come appartenenti ad un unico host, con un determinato indirizzo IP. Per ogni tipo di pod che si vuole dispiegare, lo stato desiderato viene rappresentato attraverso un *deployment*, che tra gli altri parametri permette di specificare il numero di repliche richieste. Con la sola astrazione di deployment, l'accesso alle singole pod sarebbe molto complicato, poiché sarebbe necessario conoscere l'indirizzo IP di ognuna di esse. Anche Kubernetes utilizza quindi l'astrazione di *service*, ma in questo caso assume una semantica leggermente diversa. Ogni service è dotato di una coppia IP-porta utilizzabile dai pod interni al cluster ed, eventualmente, di una coppia IP-porta raggiungibile dalla rete esterna per accedere all'applicazione. Inoltre è possibile configurare il service come load balancer per ripartire automaticamente il carico di lavoro. Un service può quindi essere considerato un *wrapper* attorno al deployment per fare in modo che le modalità di accesso ai suoi pod resti stabile anche a fronte di cambiamenti.

Infine Kubernetes introduce i concetti di *PersistenVolume* (*PV*) e *Persi-*

stentVolumeClaim (*PVC*), per garantire la persistenza dei dati. Un PV rappresenta una risorsa di storage appartenente al cluster, mentre un PVC consiste in una richiesta da parte delle pods per un certo quantitativo di memoria, di cui però non si specifica l'implementazione concreta, che verrà soddisfatta da un PV scelto da Kubernetes stesso. Le pod possono poi accedere ai PV mediante un'astrazione che permette loro di effettuarne il *mounting* sul proprio file system. Attualmente Kubernetes supporta una grande varietà di implementazioni di PV, tra cui la possibilità di sfruttare directory degli host locali o soluzioni cloud.

3.4 Docker nei processi DevOps

Con la grande diffusione di Docker come strumento per il packaging delle applicazioni, la collaborazione tra sviluppatori e operatori è stata notevolmente semplificata. I container Docker introducono un interfaccia di coordinamento tra i due settori, che abbatte il tradizionale muro tra Dev e Ops rendendo efficace la cooperazione tra essi.

Gli sviluppatori sono proprietari di codice, dipendenze e framework racchiusi nei container e si occupano di dichiarare le relazioni tra servizi tramite gli orchestratori scelti per i vari ambienti della pipeline. Gli operatori hanno invece il compito di assicurarsi del corretto funzionamento degli ambienti di produzione, avendo a che fare con aspetti di infrastruttura, scalabilità e monitoraggio con lo scopo di migliorare il servizio offerto ai clienti.

Grazie a Docker e agli strumenti di orchestrazione, il punto d'incontro tra le responsabilità dei due settori diventa drasticamente meno critico rispetto all'approccio tradizionale. Avendo un meccanismo che gestisce lo spostamento e il dispiegamento delle applicazioni si ottiene una *separation of concerns* più netta e i due team possono concentrarsi sugli aspetti che più li riguardano.

3.4.1 Workflow per l'uso di Docker nella CI/CD

Docker ha un forte impatto sulle modalità di lavoro dei due team, tanto che l'intero workflow deve essere ripensato per meglio adattarsi alla sua architettura e per integrare il suo utilizzo nella CI/CD. Nella visione di Microsoft [6], il flusso di lavoro si articola in due cicli:

- **Ciclo interno**: ha luogo nelle macchine locali dei singoli sviluppatori e va dal momento in cui si iniziano a sviluppare nuove funzionalità al momento in cui si esegue una *push* verso il server di controllo del sorgente. Il ciclo interno costituisce la prima fase del ciclo esterno.

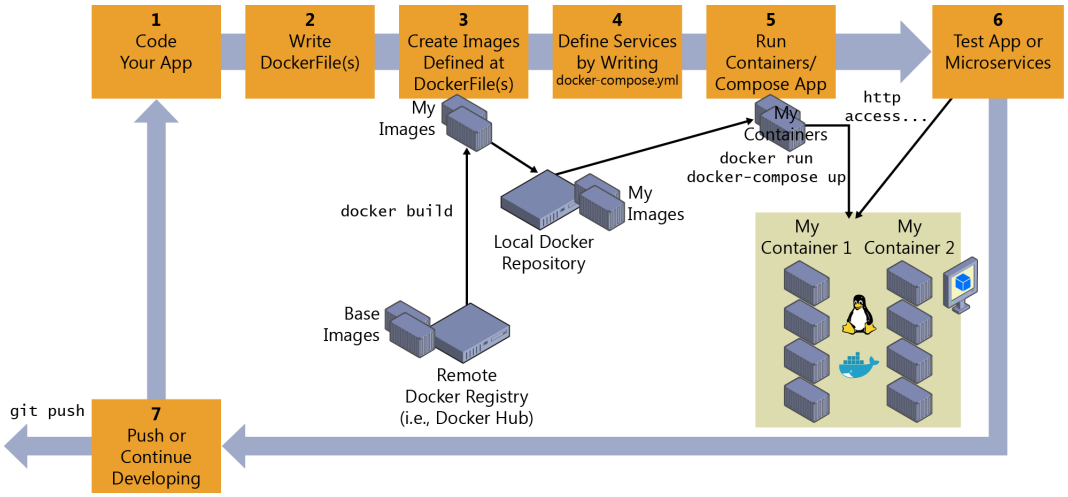


Figura 3.3: Rappresentazione del ciclo interno del flusso di lavoro secondo Microsoft

- **Ciclo esterno**: a seguito di un'operazione di push, l'applicazione viene impacchettata e attraversa la pipeline della CI/CD prima di essere messa in produzione. A questo punto gli operatori tengono sotto controllo lo stato del sistema per fornire feedback agli sviluppatori su possibili miglioramenti necessari.

In Figura 3.3 vengono schematizzati i passaggi appartenenti al ciclo interno (*inner loop*). Il processo inizia scrivendo l'applicazione, creando i Dockerfile per ciascun servizio e dichiarando le interdipendenze tra essi utilizzando il formato richiesto dall'orchestratore scelto per l'ambiente di sviluppo. È in queste fasi che il programmatore stabilisce quali saranno i contenuti di ciascuna immagine

Nel momento in cui è necessario mettere in esecuzione il sistema per eseguire dei test, si richiama l'orchestratore per creare le immagini, salvarle nel proprio repository locale e infine mettere in piedi l'architettura desiderata. Tipicamente si trae vantaggio dalla rapidità di Docker Compose nell'eseguire il deployment per testare l'applicazione multi-container sulla macchina dello sviluppatore.

Quando lo sviluppatore ritiene che l'applicazione sia stata sufficientemente testata, esegue una push delle modifiche apportate, dando inizio al ciclo esterno.

Le fasi del ciclo esterno (*outer loop*) sono riassunte in Figura 3.4. Per prima cosa, nel momento in cui il server di source-code control (SCC) riceve le nuove modifiche dallo sviluppatore, avvia il processo di integrazione indicando al server per la Continuous Integration di prelevare le modifiche più recenti. Sarà compito di quest'ultimo eseguire la build del sistema nella sua interezza,

producendo le immagini Docker che devono essere sottoposte ai test previsti dalla CI, generalmente unit tests.

Se i test hanno successo è possibile eseguire una push delle immagini Docker appena create verso un Docker Registry scelto dall'azienda per essere utilizzate nei successivi passi della Continuous Delivery ed eventualmente in produzione. Il registro diviene quindi la *fonte di verità* per tutta la CD. È importante sottolineare che le immagini che vengono sottomesse a questo registro non devono avere nulla a che fare con le immagini generate dagli sviluppatori nelle macchine locali. È per questo motivo che la pipeline viene attivata dal SCC e non direttamente dallo sviluppatore, poiché si vuole essere sicuri che le immagini utilizzino la versione più recente del codice dopo l'integrazione.

In questo momento viene attivata la Continuous Delivery. Il deployment su ciascun ambiente utilizza le immagini raccolte nel registro e può essere portato a termine con un orchestratore diverso a seconda delle necessità. La scelta viene effettuata sulla base delle caratteristiche dell'ambiente, del grado di similitudine che si vuole avere con l'ambiente di produzione e del grado di efficienza che si vuole avere. L'obiettivo è quindi quello di avere un buon compromesso tra la velocità di esecuzione dei test e il grado di confidenza con cui ci si aspetta di non avere errori in produzione.

Durante questa fase si decide quindi se l'applicazione è pronta per essere messa in produzione. Solo se i test hanno esito positivo il sistema viene dispiegato in produzione usando un orchestratore che rende semplice l'evoluzione e il monitoraggio. Disponendo di queste caratteristiche, Kubernetes fornisce un'ottima soluzione, in quanto rende molto semplice gestire la scalabilità del sistema anche senza l'intervento umano e comprende di default gran parte delle funzionalità di monitoraggio. Perciò il compito tradizionale degli operatori viene in gran parte inglobato dagli strumenti automatici, in modo che si possano focalizzare maggiormente sull'infrastruttura e sulla comunicazione del feedback agli sviluppatori.

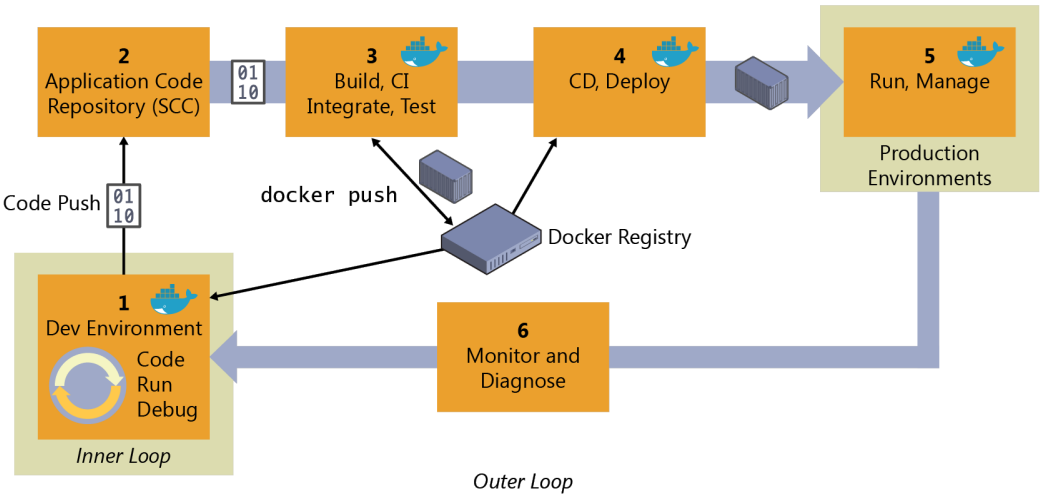


Figura 3.4: Rappresentazione del ciclo esterno del flusso di lavoro secondo Microsoft

Capitolo 4

Caso di studio

In questo capitolo si vuole esporre l'applicazione dei principi introdotti nei capitoli precedenti nel contesto del progetto Trauma Tracker. In particolare, ciò che si vuole portare a termine è uno shift del sistema in un'ottica più orientata a DevOps, automatizzando alcune di quelle procedure che venivano precedentemente eseguite manualmente. Si farà quindi leva sulla tecnologia di containerizzazione messa a disposizione da Docker, per permettere alle componenti del sistema di eseguire in maniera stabile e consistente, indipendentemente dall'ambiente fisico ospitante e in ottica di una futura implementazione di una deployment pipeline automatica. L'operatività del sistema verrà poi affidata alla potente orchestrazione di Kubernetes, per conferire ulteriore stabilità al ciclo di vita dei container, dei servizi e dell'interazione tra essi.

4.1 Il progetto Trauma Tracker

Trauma Tracker è un progetto nato con lo scopo di assistere il personale medico del trauma team di Cesena nella documentazione della *trauma resuscitation*, ovvero il processo di stabilizzazione dei parametri vitali dei pazienti soggetti a traumi di un certo livello di gravità [8]. Durante tale processo, un medico responsabile con il ruolo di *Trauma Leader* ha il compito, tra le altre cose, di stilare un *report* che riassume la successione degli avvenimenti durante la trauma resuscitation, comprendendo interventi, diagnosi, farmaci somministrati e spostamenti del paziente. Tradizionalmente, il report veniva compilato in un secondo tempo rispetto alla stabilizzazione del paziente, e costringeva il medico responsabile a basarsi sulla propria memoria e quella dei membri del team, portando quindi a una scrittura inaccurata dei documenti.

Trauma Tracker, sviluppato dai membri del PSLab dell'Università di Bologna, si pone l'obiettivo di alleviare il fardello di responsabilità di cui si deve fare carico il Trauma Leader, provvedendo a tracciare gli avvenimenti principali

sia automaticamente, tramite dispositivi smart dispiegati in maniera pervasiva all'interno dell'ospedale, sia tramite l'interazione umana *hands-free* con meccanismi di riconoscimento vocale, wearable devices, ecc. Uno tra gli obiettivi futuri del progetto è anche quello di non limitare il supporto alla semplice documentazione, ma di coadiuvare il personale nel prendere decisioni o di fornire segnalazioni sulla base dei dati forniti dal tracciamento degli avvenimenti.

4.1.1 Architettura generale e tecnologie coinvolte

Allo stato attuale, Trauma Tracker è composto da tre sottosistemi software, ognuno implementato con tecnologie diverse e con uno scopo ben preciso:

- **Trauma Tracker Infrastructure**, un backend composto da microservizi web, hostato su un server dell'ospedale.
- **Trauma Tracker Frontend o Dashboard**, un'applicazione web creata con Angular tramite la quale è possibile accedere alla *TTInfrastructure* per consultare, gestire o stampare i report, controllando insieme statistiche riguardo i traumi gestiti.
- **Trauma Leader Assistant Agent**, composto da un'applicazione Android per tablet e un'applicazione in esecuzione su smart glasses, viene usato dal Trauma Leader per descrivere in tempo reale il trauma in corso.

Tra i tre sottosistemi, quello che richiede un'analisi più dettagliata ai fini di questo progetto di tesi è senz'altro il TTInfrastructure. Come accennato, questa componente è stata ulteriormente partizionata in microservizi web. Ciascuno di essi espone un'interfaccia web basata su API HTTP di tipo REST, che utilizza Json come formato per lo scambio di dati. Il linguaggio scelto per implementare tali servizi è *Java*, sfruttando il framework *Vert.x* per gestire le richieste web. Fino ad oggi, sono stati individuati quattro tipi di microservizi:

- **Trauma Tracker Service**, col compito di mantenere l'elenco degli utenti del sistema insieme all'elenco dei report per i traumi gestiti.
- **Location Service**, addetto alla raccolta dei dati riguardo la posizione del Trauma Leader durante la trauma resuscitation. Ciò è portato a termine grazie a una serie di *rilevatori Beacon*, detti *Beacon Gateways*, posti in varie stanze dell'ospedale, che monitorano gli spostamenti dei *Beacon Tags* indossati dal Trauma Leader stesso.
- **Vital Signs Service**, incaricato di memorizzare i dati riguardo i parametri vitali dei pazienti traumatizzati, per renderli disponibili quando è necessaria la stesura dei report.

- **Trauma Stat Service**, che, sulla base dei report generati dal Trauma Tracker Service, genera delle statistiche che possono essere consultate tramite la dashboard.

Ogni servizio memorizza i dati di propria appartenenza in un database Mongo separato dagli altri, ad eccezione del servizio Trauma Stat Service, che condivide lo stesso database del Trauma Tracker Service. Naturalmente, l'attuale implementazione non è accettabile in un'architettura a microservizi, poiché viola il principio di incapsulamento (Sezione 2.2.2). Sarà quindi necessaria, anche se non effettivamente implementata per ciò che concerne questo lavoro di tesi, la progettazione di una strategia che risolva tale situazione.

4.2 Analisi e progettazione del deployment

4.2.1 Problematiche dello stato attuale

Al momento, Trauma Tracker è già in fase di sperimentazione all'interno dell'ospedale Bufalini di Cesena. Tuttavia, la natura manuale di gran parte dei processi, in particolare il deployment, la rilevazione di malfunzionamenti dell'infrastruttura e la tempestiva risposta ad essi, rendono la gestione degli aspetti operativi frustrante, sia per gli utilizzatori del sistema, sia per i responsabili di tali operazioni.

Obiettivo principale di questo lavoro di tesi è quello di automatizzare tali processi, utilizzando i container Docker per semplificare la creazione e l'integrazione di nuove istanze di servizi e sfruttando le potenzialità offerte da Kubernetes per monitorare lo stato di ciascuna di esse e reagendo automaticamente ai malfunzionamenti. Nonostante allo stato attuale il sistema abbia a disposizione una sola macchina fisica su cui eseguire, si cercherà di creare un'infrastruttura che non dipenda da tale limitazione, in modo da rendere l'aggiunta di nuovi nodi fisici al cluster di Kubernetes un'operazione di costo minimo.

Trauma Tracker è allo stato attuale dispiegato su un unico *server Linux* collocato nella struttura dell'ospedale. Al suo interno sono installate tutte le applicazioni di backend che compongono il sistema, in particolare i database, i microservizi e il server che distribuisce ai client la Trauma Tracker Dashboard. Per tale motivo, gli applicativi sono stati sviluppati dando per scontato di avere tutte le dipendenze in esecuzione sullo stesso host. La transizione verso l'utilizzo di Kubernetes, che segue la filosofia secondo la quale ogni servizio esegue su un diverso container, richiede quindi che le comunicazioni con altre componenti venga effettuata tramite scambio di messaggi di rete. Questo significa che, nella ristrutturazione del progetto al fine di essere eseguito su

un cluster Kubernetes, sarà necessario aggiustare le configurazioni per fare in modo che le richieste vengano effettuate verso gli indirizzi e le porte corretti, piuttosto che verso quelle del *localhost*.

4.2.2 Ristrutturazione dell'architettura

Per meglio avvicinarsi ai principi dell'architettura a microservizi, ma soprattutto per adattarsi all'architettura di Kubernetes, il primo obiettivo è quello di riconfigurare la struttura con cui Trauma Tracker viene dispiegato.

Al momento, le assunzioni fatte per via della disponibilità di un'unica macchina hanno reso le componenti del sistema fortemente accoppiate, seppur indirettamente. Per esempio, è sufficiente pensare alla convivenza dei database, seppur indipendente, sotto un solo servizio Mongo installato sul server. Una situazione del genere crea degli accoppiamenti impliciti tra i servizi, che in casi particolari possono portare effetti indesiderati. La ristrutturazione prevede quindi che ciascun database venga completamente isolato in un container separato, su cui esegue una istanza di Mongo indipendente rispetto alle altre.

La Figura 4.1 mostra la versione attuale del deployment, con ciascun applicativo (nell'immagine denotato con un cerchio) installato su un unico server. Ciascun applicativo viene acceduto sia dai client, sia dagli altri applicativi appartenenti all'host stesso, tramite delle porte su cui essi sono in ascolto, evidenziate nello schema.

L'architettura di Kubernetes richiede che venga posta attenzione agli end-point che devono essere esposti all'esterno e all'interno del cluster. Kubernetes distingue il modo con cui un applicazione può essere contattata a seconda che il client si trovi all'interno o all'esterno del cluster, rendendo quindi necessario sia definire i nuovi end-point, sia riconfigurare le applicazioni per far fronte a tale cambiamento.

La soluzione raggiunta è mostrata in Figura 4.2. Si osservi che i cerchi assumono ora il significato di Service, così come definito dall'architettura di Kubernetes (Sezione 3.3.3). In quanto tali, uno tra gli end-point messi a disposizione può essere utilizzato esclusivamente da altri componenti del cluster, ma non da client esterni. Per questo tipo di end-point sono state mantenute le porte originali, in modo da minimizzare le riconfigurazioni dei servizi. Mentre gli end-point interni al cluster sono sufficienti per garantire l'accesso alle istanze di Mongodb da parte dei quattro microservizi, per le richieste che arrivano da client esterni è necessario che i microservizi e il frontend mettano a disposizione una coppia IP-porta pubblica. Kubernetes pone una limitazione sul range di porte utilizzabile pubblicamente e si dovrà perciò rinunciare a mantenere le porte scelte in origine che verranno mappate su porte con valore oltre 30000.

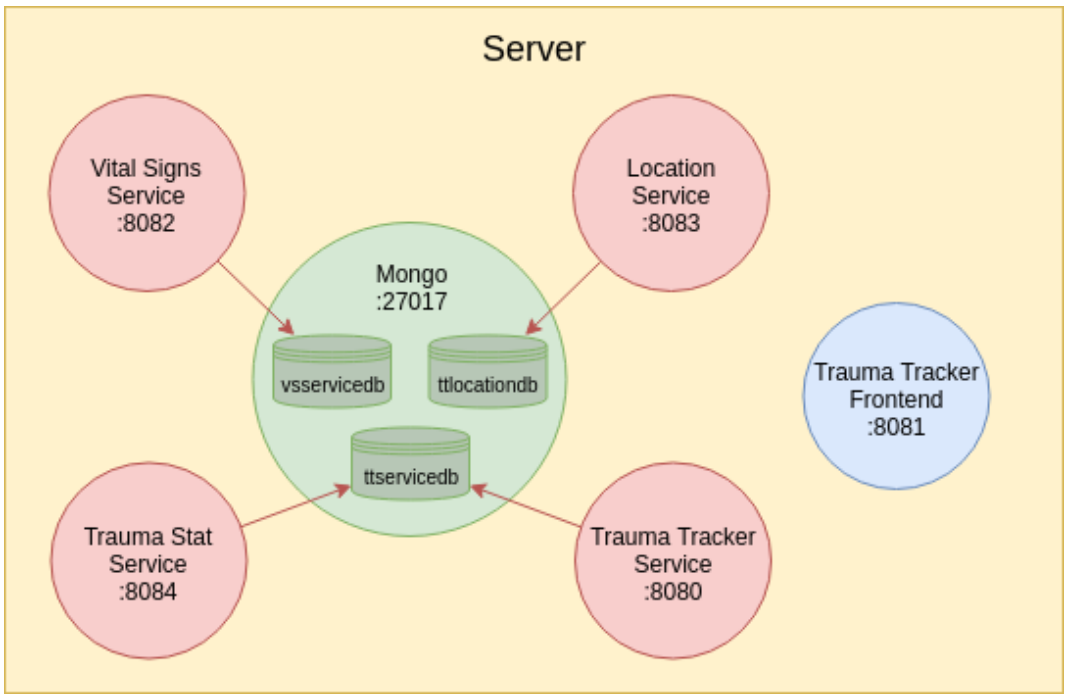


Figura 4.1: Schema del deployment attuale per il backend di Trauma Tracker, dispiegato su un'unica macchina fisica

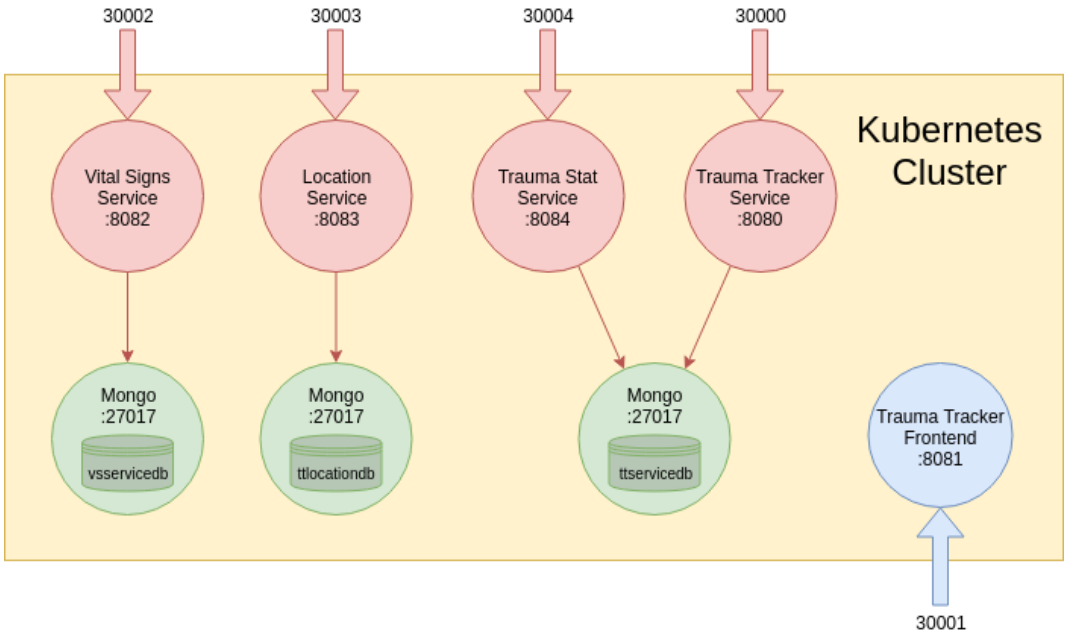


Figura 4.2: Rifattorizzazione dell'architettura di Trauma Tracker per essere dispiegata su un cluster Kubernetes

4.3 Containerizzazione dei servizi

Una volta progettata la nuova infrastruttura per l'ecosistema Trauma Tracker, il passo successivo è quello di definire i contenuti e il comportamento delle immagini da cui verranno derivati i container. Come già anticipato nella Sezione 3.2.2, Docker mette a disposizione degli sviluppatori la possibilità di dichiarare le istruzioni che vanno a formare l'immagine tramite i Dockerfile. Ogni istruzione corrisponderà ad un *layer immutabile* dell'immagine finale.

Trauma Tracker richiede che vengano dispiegate tre tipologie di container che vengono eseguite come applicazioni server-side:

- **I microservizi**, che richiedono che all'interno del container sia presente l'ambiente di esecuzione Java (JRE versione 8).
- **Il database**, che richiede l'esecuzione del servizio Mongo.
- **Il frontend**, per permettere agli utenti di accedere alla Dashboard di TraumaTracker come applicazione web.

Si osservi che in alcuni casi ciascuna tipologia potrebbe essere associata a due immagini, una utilizzata dagli sviluppatori sulla propria macchina e una destinata ad essere usata in produzione (e quindi all'interno della pipeline). Tale differenziazione è presente principalmente per permettere agli sviluppatori di fare affidamento agli strumenti di debugging offerti dai linguaggi e dalle piattaforme. Nel caso di Trauma Tracker, questa opportunità è stata sfruttata per il *TraumaTrackerFrontend*, per avere a disposizione gli strumenti di debugging offerti da Angular come l'*hot reloading*.

Nella creazione delle immagini è importante seguire delle linee guida che hanno lo scopo di minimizzarne le dimensioni in termini di memoria e di massimizzare la predicibilità del risultato finale.

La prima richiede che le immagini che si utilizzano come base di partenza siano, se presenti, quelle messe a disposizione nei repository pubblici, costruite appositamente per essere utilizzate come immagini base. Solitamente queste immagini vengono create ad-hoc per soddisfare un certo tipo di necessità, che sia una piattaforma di esecuzione, un insieme di librerie o addirittura un applicazione pronta all'esecuzione. In sostanza, si vuole evitare di reinventare la ruota continuamente, poiché spesso questo porta a immagini troppo pesanti, contenenti risorse inutilizzate e dal comportamento impredicibile.

Una seconda regola prevede che anche la compilazione dei sorgenti venga effettuata all'interno dei container, per cercare di annullare le dipendenze con la macchina host fisica. L'adozione di questa pratica però genererebbe immagini finali inquinate dalla presenza dei sorgenti e delle risorse di compilazione,

che sarebbero poi inutilizzate a run-time, occupando ulteriore spazio su disco. Docker ha posto rimedio a questo con le cosiddette *multi-stage builds*: esse consistono nella suddivisione della costruzione in più stadi, tipicamente uno per la compilazione e uno per l'eseguibile finale, ognuno dei quali genera un'immagine a sé stante. Ogni stage può utilizzare una diversa immagine di partenza e prelevare file dagli stadi precedenti. In questo modo è semplice definire uno stadio in cui vengono inseriti i sorgenti e su cui viene eseguita la compilazione ed uno stadio che preleva dal precedente solo gli artefatti necessari a run-time. Spesso si fa riferimento a questa tecnica con il nome di *builder pattern*.

Infine è buona norma, quando possibile, ordinare le istruzioni in ordine crescente di frequenza di cambiamento. Questo è importante in quanto Docker effettua un pesante caching dei layer delle immagini che ha già costruito, in modo che le successive build non vengano rieseguite da capo ogni volta, ma riutilizzino quando possibile layer già creati. Ponendo per prime le istruzioni più stabili si favorisce il riuso dei layer e si rende il processo di costruzione molto più rapido.

Il perseguimento delle linee guida sopracitate ha portato, per quanto riguarda i microservizi, alla stesura di un Dockerfile simile a quello riportato nel Listato 4.1. Esso contiene le istruzioni per creare l'immagine per il *TraumaTrackerService* ed è sostanzialmente identico a quelli utilizzati dagli altri servizi, se non per la differenza nella porta esposta e nella nomenclatura dei file di configurazione.

```
# Stage 1 -- build the .jar using gradle
FROM gradle:4.8-jdk8-alpine AS builder
COPY --chown=gradle:gradle . /home/gradle/app
WORKDIR /home/gradle/app
RUN gradle shadowJar --no-daemon

# Stage 2 -- copy the .jar from the builder and run it
FROM openjdk:8-jre-slim
EXPOSE 8080
WORKDIR /app
COPY --from=builder /home/gradle/app/build/libs/ .
COPY --from=builder /home/gradle/app/config.json .
ENTRYPOINT java -jar *.jar
```

Listato 4.1: Dockerfile utilizzato per la creazione dell'immagine per il TraumaTrackerService

Analizzando il listato, si osserva che segue il builder pattern, in quanto è

suddiviso in due stadi. Nel primo, denominato *builder* e derivato dall'immagine ufficiale di Gradle (*gradle:4.8-jdk8-alpine*), viene effettuata la copia dei sorgenti dall'host all'immagine assegnando *gradle* come proprietario dei file. Infine viene generato il .jar eseguibile eseguendo il comando di compilazione nella directory dove era stato copiato il codice.

Il secondo stadio viene invece basato sull'immagine ufficiale *openjdk:8-jre-slim*, contenente solo l'ambiente d'esecuzione Java. In questo nuovo stage è necessario impostare i parametri per l'esecuzione corretta del servizio: viene quindi esposta la porta 8080 e vengono copiati dallo stadio precedente il .jar e il file di configurazione del servizio. Infine viene dichiarato il comando (*ENTRYPOINT*) da eseguire una volta messa in esecuzione l'immagine, ovvero l'esecuzione del servizio.

Per il TraumaTrackerFrontend, come anticipato, sono stati concepiti due diversi Dockerfile. Il primo (Listato 4.2) è stato progettato per essere utilizzato in fase di sviluppo per testare l'applicazione e il suo comportamento all'interno del container, sfruttando allo stesso tempo i tool di debugging di Angular.

```
FROM node:12.2.0
RUN npm install -g @angular/cli@7.3.9
VOLUME /app
WORKDIR /app
EXPOSE 8081
CMD ng serve --host 0.0.0.0 --port 8081
```

Listato 4.2: Dockerfile utilizzato per la creazione dell'immagine per il TraumaTrackerFrontend in fase di sviluppo

In questo caso i sorgenti vengono passati al container tramite il meccanismo dei Docker Volumes. Nel Dockerfile si dichiara infatti che la directory */app* sarà, al momento del lancio del container, mappata direttamente a una dirctory dell'host, in particolare la directory contenente il progetto Angular. Questa scelta comporta un'importante assunzione, ovvero che l'host che lancia il container contenga in una sua directory i sorgenti del progetto. Mentre questa assunzione è totalmente rispettata se si mette in esecuzione l'immagine dalla macchina dello sviluppatore, essa perde di significato nel momento in cui ci si sposta sui server dell'ambiente di produzione. In tal caso è quindi necessario che i file necessari siano direttamente copiati all'interno dell'immagine (Listato 4.3).

```
FROM node:12.2.0
WORKDIR /app
RUN npm install -g @angular/cli
EXPOSE 8081
```

```
COPY . .
CMD ng serve --host 0.0.0.0 --port 8081
```

Listato 4.3: Dockerfile utilizzato per la creazione dell'immagine per il TraumaTrackerFrontend in ambiente di produzione

Infine, per il database Mongodb, l'immagine finale è quasi identica a quella ufficiale fornita tramite Docker Hub. L'unica aggiunta è uno script bash che verrà utilizzato dal controllore di Kubernetes per monitorare lo stato di attività del container, al fine di assicurarsi l'effettiva operatività del DBMS.

Dopo aver definito i Dockerfile necessari, sarà necessario procedere con l'effettiva creazione delle Docker images. Nonostante l'operazione sia facilmente eseguibile sfruttando il comando *docker build*, costruendo singolarmente ogni immagine quindi, è possibile affidarsi a una funzionalità di Docker Compose (il comando *docker-compose build*) per eseguire con un singolo comando la build di tutti gli artefatti.

Avendo a disposizione tutte le immagini necessarie, si può ora procedere con la dichiarazione delle risorse di Kubernetes.

4.4 Automazione del deployment con Kubernetes

Questa fase del processo richiede che vengano stabilite le caratteristiche e il comportamento delle risorse che si vogliono facciano parte del cluster. Kubernetes prevede la stesura di file manifest, scritti in formato YAML, tramite i quali si specificano le proprietà di tali risorse e che vengono successivamente sottomessi, col comando *kubectl apply*, al Master del cluster che provvederà ad applicarli effettivamente.

Si tenga a mente che, al fine di progettare e testare l'infrastruttura di Kubernetes, si è scelta una soluzione gratuita e con scopo di apprendimento, quindi non adatta a essere utilizzata in produzione: *Minikube*. Minikube permette la creazione di un cluster Kubernetes minimale, composto da un solo nodo implementato da una macchina virtuale creata su *Virtual Box*. La configurazione di questa macchina, in particolar modo la quantità di memoria centrale e di spazio su disco, è personalizzabile per far fronte al carico di lavoro che ci aspetta di mettere in esecuzione. Nel caso di questo progetto, che richiede delle capacità computazionali e di storage superiori a quelle di default, si è visto che le specifiche minime per un funzionamento corretto sono:

- 4GB di memoria centrale.
- 60GB di spazio su disco.

Tenendo a mente i principali tipi di risorse descritti nella Sezione 3.3.3 e l'architettura finale riassunta dalla Figura 4.2, si andranno ad analizzare gli oggetti Kubernetes necessari.

4.4.1 Persistenza dei dati

Per prima cosa, è indispensabile, al fine di mantenere la persistenza dei dati salvati dai database Mongodb, definire dei volumi su cui salvare i documenti. La separazione tra PV e PVC messa in atto da Kubernetes dà la possibilità di disaccoppiare l'utilizzo dei volumi dalla loro implementazione effettiva. Ciò è stato sfruttato nella scelta del tipo di PV utilizzato per Trauma Tracker. Tipicamente, nell'ambito di applicazioni dispiegate su cluster con molteplici nodi, in cui i processi possono essere trasferiti da una macchina all'altra, si sceglie di affidarsi a servizi di storage basati sul cloud, in modo da non accoppiare il salvataggio dati con uno specifico host. Data la situazione attuale di Trauma Tracker, che vede la disponibilità di una sola macchina, si può pensare di limitarsi a un semplice mapping sulle directory dell'host fisico, senza il rischio che il re-scheduling delle pod di mongo comporti una perdita di dati. Qualora in futuro le disponibilità dell'infrastruttura evolvessero, sarebbe sufficiente aggiornare i PV creati per utilizzare possibili soluzioni cloud senza il bisogno di modificare altre parti del sistema. Le impostazioni utilizzate per configurare i PV e i PVC associati al Trauma Tracker Service sono mostrate nei Listati 4.4 e 4.5.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodata-tt-pv
  labels:
    type: local
    app: tt
spec:
  storageClassName: manual
  capacity:
    storage: 4Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/mongo_data/tt"
```

Listato 4.4: Definizione del PV utilizzato dal database associato al Trauma Tracker Service

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodata-tt-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 4Gi
  selector:
    matchLabels:
      app: tt
```

Listato 4.5: Definizione del PVC utilizzato dal database associato al Trauma Tracker Service

4.4.2 Definizione dello stato desiderato

Grazie all'astrazione di deployment, Kubernetes lascia la possibilità agli amministratori del sistema di decidere in che modo debbano essere dispiegate le componenti software. In sostanza, con un deployment si decidono due caratteristiche di ciascuna componente: il *template* da cui generare le pod e il numero di repliche da istanziare.

Andando più nello specifico, il template permette di dichiarare, oltre ai container adibiti a mantenere l'applicativo vero e proprio, un altro tipo di container che nel gergo di Kubernetes viene chiamato *init container*. Ogni pod, prima di lanciare i container standard, esegue fino alla terminazione tutti gli init container. Per questo motivo, vengono tipicamente utilizzati per attendere la disponibilità di altri servizi che costituiscono dipendenze di start-up della pod stessa. Nel caso di Trauma Tracker, gli init container sono stati utilizzati per effettuare l'attesa da parte dei servizi per l'operatività del database.

Inoltre il template permette di configurare la modalità con cui si decide se un container è funzionante o meno, attraverso le cosiddette *readiness e liveness probes*. Il controllore di Kubernetes effettua periodicamente sui container delle interrogazioni, definite dall'amministratore o dallo sviluppatore, per decidere lo stato di attività della pod stessa, con lo scopo di disattivarle o ricrearle a fronte di malfunzionamenti.

Un esempio di definizione di deployment utilizzato per il Trauma Tracker Service può essere consultato nel Listato 4.6.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ttservice-deployment
  labels:
    app: tt
    tier: backend
spec:
  selector:
    matchLabels:
      app: tt
  replicas: 1
  template:
    metadata:
      labels:
        app: tt
        tier: backend
    spec:
      containers:
      - name: ttservice
        image: ttservice:prod
        ports:
        - containerPort: 8080
        readinessProbe:
          httpGet:
            path: /gt2/traumatracker/api/version
            port: 8080
          initialDelaySeconds: 30
          periodSeconds: 20
      initContainers:
      - name: init-wait-mongo
        image: busybox
        command:
        - "/bin/sh"
        - "-c"
        - "until nslookup tt-mongo; do sleep 10; done;"
```

Listato 4.6: Definizione del deployment utilizzato dal Trauma Tracker Service

Gli altri deployment, compreso quello per il frontend, sono riconducibili a quello mostrato e vengono omessi.

Per applicazioni di tipo stateful, ad esempio i database, utilizzare i deployment può portare a incongruenze nel momento in cui una pod dovesse essere ri-schedulata. Con i deployments, lo spostamento di una pod tra un nodo e un altro viene fatto semplicemente distruggendola e ricreandola sul nodo destinazione. Questo comportamento non è adatto alle applicazioni che mantengono in qualche modo uno stato e deve quindi essere modificato. Kubernetes introduce gli *Stateful Sets*, che risolvono proprio questo problema. Per Trauma Tracker, i database sono stati quindi dispiegati come Stateful Sets, la cui definizione è riportata nel Listato 4.7.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: tt-mongo-statefulset
  labels:
    app: ttmongo
    tier: db
spec:
  serviceName: tt-mongo
  replicas: 1
  selector:
    matchLabels:
      app: ttmongo
  template:
    metadata:
      labels:
        app: ttmongo
        tier: db
    spec:
      containers:
      - name: tt-mongo
        image: ttmongo:prod
        ports:
        - containerPort: 27017
        volumeMounts:
        - name: mongo-volume
          mountPath: /data/db
        readinessProbe:
          exec:
```

```
          command:
          - "/bin/check_readiness.sh"
        initialDelaySeconds: 10
        periodSeconds: 20
    volumes:
    - name: mongo-volume
      persistentVolumeClaim:
        claimName: mongodata-tt-pv-claim
```

Listato 4.7: Definizione dello Stateful Set utilizzato per il database associato al Trauma Tracker Service

In questo file si può osservare anche l'utilizzo del PVC definito in precedenza, che viene montato nella directory */data/db*, ovvero la directory in cui di default mongo salva i dati persistenti.

4.4.3 Esposizione dei servizi

Una volta definito lo stato desiderato del sistema, il passaggio successivo richiede di unificare tutte le pod appartenenti a un deployment o a uno stateful set sotto un'unica coppia di end-point, uno dei quali utilizzabile stabilmente dall'intero cluster e uno dai client esterni. Occorre quindi definire un Kubernetes Service per ciascuna componente appartenente al sistema.

Kubernetes consente di scegliere tra varie tipologie di service, ognuna adatta a fornire diversi tipi di funzionalità. La tipologia scelta per i servizi accessibili dall'esterno del cluster è *NodePort*, che consente di bilanciare il carico di richieste tra tutte le pod appartenenti al servizio stesso e residenti su uno stesso nodo. Questa tipologia è allo stato attuale sufficiente poiché il cluster disponibile è composto da un solo nodo. Qualora le dimensioni crescessero, sarebbe sufficiente spostarsi verso un servizio di tipo LoadBalancer, che gestisce anche la possibilità che un servizio sia dispiegato su più nodi.

Come già accennato, un service di tipo NodePort è associato a due endpoint configurabili dall'utente tramite il file YAML. Nel Listato 4.8 si può osservare la definizione del service associato al Trauma Tracker Service che, secondo lo schema definito in Figura 4.2, effettua il forwarding dalla porta 8080 del service verso la porta 8080 della pod per le richieste provenienti da dentro il cluster e dalla porta 30000 del nodo alla porta 8080 della pod per le richieste esterne.

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: ttservice
  labels:
    app: tt
    tier: backend
spec:
  ports:
  - port: 8080
    targetPort: 8080
    nodePort: 30000
  selector:
    app: tt
  type: NodePort
```

Listato 4.8: Definizione del service associato al Trauma Tracker Service

Per i database mongo, non accessibili direttamente dall'esterno del cluster, è sufficiente la tipologia ClusterIP, che pubblica un unico end-point visibile solo dai nodi del cluster.

4.5 Analisi dei risultati raggiunti

La nuova versione del deployment di Trauma Tracker genera un forte impatto sul sistema, che vede modificati diversi aspetti del suo ciclo di vita. Queste novità, benché migliorino notevolmente il workflow sia dal punto di vista dello sviluppo vero e proprio, sia per quanto riguarda aspetti trasversali come monitoraggio e operatività del sistema, lasciano aperte delle nuove sfide di cui ci si dovrà fare carico.

L'automazione delle procedure delicate come il deployment e la reazione ai guasti, rappresentano solo il primo passo verso una completa applicazione delle pratiche DevOps, nonostante costituiscano una prerogativa non sottovalutabile.

4.5.1 Benefici ottenuti

L'uso di Docker e Kubernetes nel contesto di Trauma Tracker ha portato evidenti benefici in diversi aspetti del sistema e del suo ciclo di vita. Primo fra tutti è la completa e rigorosa *astrazione* di ogni aspetto del deployment, che porta il sistema a non essere più strettamente legato all'infrastruttura correntemente a disposizione nell'ospedale. Gli sviluppi futuri per il progetto prevedono l'aggiunta di svariate funzionalità, che potrebbero richiedere grosse ristrutturazioni della rete, la necessità di nuove macchine o la modifica del-

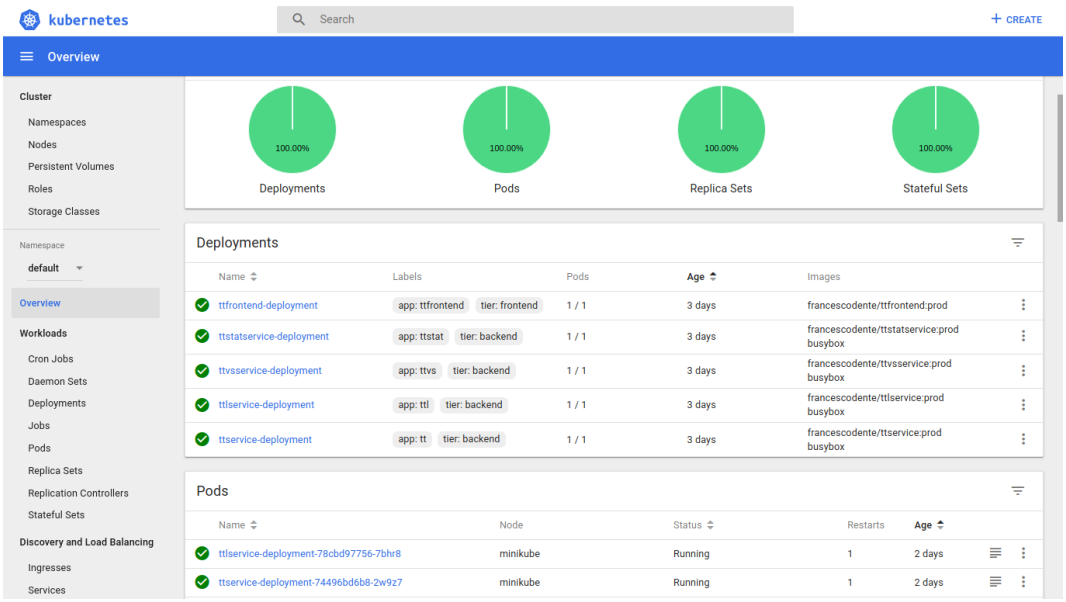


Figura 4.3: Visualizzazione dello stato globale del cluster tramite la dashboard operativa di Kubernetes

l'infrastruttura su cui attualmente è installato l'ecosistema. L'utilizzo di una modalità di deployment e di aggiornamento fragile come quella attualmente in vigore, potrebbe quindi rendere l'evoluzione del sistema esageratamente macchinosa, tanto da convogliare la maggior parte del lavoro su aspetti di scarsa rilevanza ai fini dell'effettivo sviluppo. Con le tecnologie adoperate questi problemi vengono in gran parte superati, poiché i servizi coinvolti si basano su un'astrazione dell'infrastruttura in parte indipendente da quella sottostante.

In secondo luogo, subiscono una forte spinta in positivo anche il monitoraggio dello stato operativo del sistema e la conseguente reazione ai guasti, punto cardine dei principi DevOps, ma pressoché inesistente al momento. Nella situazione attuale, i malfunzionamenti di un servizio vengono principalmente individuati dall'utente finale, il quale si vede spesso costretto a contattare i responsabili del progetto per segnalare l'inattività del sistema. Kubernetes aiuta in diversi modi a far fronte a queste eventualità, sia grazie alla funzionalità di riavvio automatico dei servizi, sia grazie alla sua dashboard di controllo (Figura 4.3). In particolare, grazie a quest'ultima è possibile avere una visione d'insieme dello stato del cluster e, eventualmente, di avere informazioni dettagliate su performance, log e statistiche riguardanti i singoli servizi. Avere a disposizione questi strumenti è fondamentale ai fini di garantire la tracciabilità degli errori e consente di rilevare i difetti nel codice in tempi più ristretti.

Infine, la ristrutturazione architetturale messa in atto porterà a una mag-

giore indipendenza tra i servizi già presenti, uniformandosi maggiormente ai principi dei microservizi. Di conseguenza, l'aggiunta di nuove funzionalità, l'eventuale aggiornamento di quelle esistenti o un semplice miglioramento delle performance, richiederà meno coordinazione e impatterà in maniera decisamente ridotta sulla restante parte del sistema.

4.5.2 Sfide e sviluppi futuri

Il lavoro svolto per questo progetto di tesi lascia alcune possibilità aperte per un futuro miglioramento del sistema Trauma Tracker.

La prima tra esse è la possibile adozione della filosofia *Continuous*, che consentirebbe l'implementazione di una pipeline automatizzata basata sull'uso di Docker come strumento di packaging e di Kubernetes al fine di rendere automatica la procedura di provisioning di ciascun ambiente. Si renderebbe in questo modo l'aggiunta di nuove funzionalità un processo molto più frequente e affidabile rispetto ad ora, e permetterebbe agli sviluppatori di aggiungere nuove funzionalità al sistema con maggiore confidenza. Questo richiederebbe, oltre alla definizione della pipeline in sé, la scrittura di test automatici da eseguire durante i vari stadi e l'approvvigionamento di un CI server, insieme a un Docker Registry in cui memorizzare le immagini per la Continuous Delivery.

Inoltre, ricollegandosi alla problematica anticipata alla fine della Sezione 4.1.1, un futuro oggetto di analisi potrebbe essere lo studio di una nuova strategia di condivisione di dati tra il Trauma Tracker Service e il Trauma Stat Service, al momento in totale contrasto con i principi dei microservizi. Una simile dipendenza, in un contesto in forte evoluzione come Trauma Tracker, potrebbe avere effetti negativi sui futuri sviluppi per il progetto e ritardare ulteriormente la consegna delle nuove funzionalità che ci si augura di offrire all'ospedale. La soluzione più indicata secondo la letteratura richiede che il servizio non proprietario dei dati, in questo caso il Trauma Stat Service, acceda ad essi indirettamente, quindi tramite l'interfaccia pubblica del Trauma Tracker Service o tramite strategie basate su eventi. Questa decentralizzazione renderebbe poi necessaria un'ulteriore attenzione alla possibilità di generare inconsistenze nei dati e renderebbe necessaria l'implementazione di un meccanismo di eventual consistency, probabilmente basato sul concetto di Saga (Sezione 2.5.2).

Conclusioni

A seguito lavoro svolto all'interno del contesto di Trauma Tracker, ciò che emerge principalmente è la grande quantità di competenze richieste per poter affermare di seguire appieno lo stile DevOps, sia negli ambiti puramente implementativi, sia per gli aspetti più trasversali. Mentre il coinvolgimento delle pratiche della sua cultura possa rappresentare un costo non esageratamente eccessivo nel caso sia già prevista all'avvio di un progetto, si è palesato come effettuare il passaggio a sviluppo già inoltrato possa costituire uno scoglio non facile da sormontare.

Il lavoro svolto per avvicinare Trauma Tracker alla cultura DevOps, seppur portato a termine a seguito di uno studio approfondito delle problematiche presenti nello stato attuale del sistema, rappresenta solo il punto di partenza per poter mettere in pratica i principi citati nei primi capitoli. I benefici sull'agilità nello sviluppo messe in gioco da DevOps, dai microservizi e dall'uso di tecnologie di supporto devono quindi essere sostenute da un forte cambio di mentalità, senza il quale i vantaggi promessi sarebbero totalmente annullati. Tuttavia non si può negare che, con una corretta e completa applicazione dei principi, i vantaggi che ne derivano siano innegabili.

Da un punto di vista più tecnico, l'unico tipo di valutazione che si è potuta eseguire riguardo il contributo fornito è stata di tipo prettamente qualitativo, per via dell'utilizzo di Minikube come soluzione per la gestione del cluster Kubernetes. Effettuare misurazioni su tempi e performance in tale ambiente sarebbe stato poco veritiero, poiché Minikube ignora gran parte degli aspetti critici presenti in produzione. Nonostante ciò, l'obiettivo che ci si era preposti è stato portato a termine e il sistema esegue correttamente e stabilmente nell'ambiente di prova utilizzato.

La soluzione raggiunta può quindi essere considerata efficace sia dal punto di vista architetturale, poiché risolve la maggior parte delle violazioni dei principi presenti in precedenza, e dal punto di vista implementativo, in quanto è stata resa sufficientemente generale da consentire l'adattamento a una futura espansione del prodotto. Quest'ultima proprietà è stata infatti, per tutta la durata del progetto, il punto primario da conseguire, per via della limitata conoscenza del futuro di Trauma Tracker e dei possibili miglioramenti

dell'infrastruttura fisica.

In conclusione, il mantenimento e la conformazione degli sviluppi futuri del sistema ai concetti di DevOps e dell'architettura a microservizi, insieme alla risoluzione delle eventuali problematiche ancora presenti, saranno una prerogativa per il successo dei futuri aggiornamenti.

Ringraziamenti

Desidero innanzitutto ringraziare i relatori di questa tesi, Alessandro Ricci e Angelo Croatti, per la loro disponibilità e la passione con cui svolgono il loro lavoro, grazie ai quali posso dire di essere molto più che soddisfatto dei risultati ottenuti.

Un grazie di cuore alla mia famiglia, per la grande quantità di supporto sotto ogni punto di vista e senza la quale non avrei mai potuto realizzare questo traguardo.

Ringrazio anche tutti i miei gruppi di amici, colleghi universitari e compagni di squadra per essere stati vicino a me in questi mesi e per essere passati sopra alle mie assenze. Vi prometto che mi farò perdonare.

Infine ringrazio la pallavolo per essere stata la mia fedele valvola di sfogo quando l'ansia e lo stress si accumulavano, ma soprattutto per avermi dato delle splendide amicizie.

Bibliografia

[1] Amazon AWS. Cos'è l'integrazione continua? https://aws.amazon.com/it/devops/continuous-integration/.

[2] Len Bass, Ingo Weber, and Liming Zhu. *DevOps, A Software Architect's Perspective*. Pearson, 2015.

[3] Martin Fowler. Bounded context, 2014. https://martinfowler.com/bliki/BoundedContext.html.

[4] Docker Inc. Docker overview. https://docs.docker.com/engine/docker-overview/.

[5] Microsoft. Communication in a microservice architecture, 2018. https://docs.microsoft.com/it-it/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.

[6] Microsoft. Containerized docker application lifecycle with the microsoft platform and tools, 2019. https://docs.microsoft.com/it-it/dotnet/architecture/containerized-lifecycle/index.

[7] Sam Newman. Principles of microservices, 2015. https://www.youtube.com/watch?v=PFQnNFe27kU&t=2480s.

[8] Alessandro Ricci, Angelo Croatti, and Sara Montagna. A personal medical digital assistant agent for supporting human operators in emergency scenarios.

[9] Chris Richardson. Microservices + events + docker = a perfect trio, 2016. https://www.youtube.com/watch?v=sSm2dRarhPo.