

UNIVERSITÀ DEGLI STUDI DI UDINE

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Corso di Laurea Triennale in Informatica

Tesi di Laurea

INTEGRATION TESTING E DEVOPS: SVILUPPO E SPERIMENTAZIONE DELLA LIBRERIA KUBESUITE

Relatore:
Prof. TASSO CARLO

Laureando:
BELLIATO RICCARDO

Correlatore:
MAIONE ROSARIO

ANNO ACCADEMICO 2020-2021

Introduzione

DevOps, microservizi e Serverless computing sono alcune delle ultime tendenze nell'ambito dell'Ingegneria del Software.

L'azienda Beantech di Udine ha deciso di investire molte risorse sulla transizione verso la metodologia di sviluppo DevOps e le architetture a microservizi.

In particolare si è cercato di applicare questi principi allo sviluppo del prodotto Brainkin, di cui una parte è stata implementata utilizzando due particolari tecnologie: Azure Functions (un framework di Serverless Computing sviluppato da Microsoft) e Kubernetes (un sistema di orchestrazione di container).

Uno dei problemi rilevati nello sviluppo delle Azure Function per Brainkin è stata la carenza di una fase di testing strutturata e aderente alle specifiche della metodologia DevOps.

Lo scopo del progetto presentato in questa tesi è quello di realizzare degli strumenti di supporto al testing automatico delle Azure Functions in esecuzione su Kubernetes, che si concretizzeranno nell'implementazione di una libreria di supporto al testing e la descrizione di una sequenza di istruzioni (pipeline) standard che esegue l'intero processo su Azure DevOps.

I primi capitoli saranno fortemente introduttivi rispetto ai concetti e le tecnologie utilizzate nella realizzazione del progetto, con particolare focus su che cos'è DevOps, cos'è un container, come funzionano Kubernetes e Azure Functions. A seguire un breve capitolo di presentazione del prodotto Brainkin.

Terminata la parte introduttiva è presente un capitolo nel quale verranno analizzati contesto e requisiti del progetto, seguiti dalla progettazione architetturale e dalla discussione di alcune scelte implementative adottate.

A conclusione del tutto, è stato inserito un capitolo nel quale verranno analizzati i pattern e le metodologie di testing adottate per le Azure Functions, insieme a due esempi di test funzionanti.

Indice

Introduzione	iii
Elenco delle figure	ix
Elenco delle tabelle	xi
1 Modelli e tecnologie di sviluppo agili per le architetture a microservizi	1
1.1 DevOps e Continuous Practices (CP)	1
1.1.1 I 4 concetti fondamentali di DevOps	3
1.1.2 Implementare DevOps: quattro approcci	4
1.1.3 Implementare DevOps: le Continuous Practices	6
1.1.4 Le <i>pipelines</i>	7
1.1.5 Ostacoli e pratiche comuni nell'adozione di DevOps e delle CP	8
1.1.6 Approfondimento: Azure DevOps	10
1.2 Architetture a microservizi e DevOps	13
1.2.1 Containerizzazione: Docker	15
1.2.2 Orchestrazione di container: Kubernetes	18
1.2.3 Panoramica delle API di Kubernetes	19
1.2.4 Serverless computing	21
1.3 Testing di microservizi e framework automatici di test	23

2	Sistemi IoT per il monitoraggio di una linea di produzione: il prodotto Brainkin	25
2.1	Interfaccia e funzionalità di Brainkin	25
2.2	Azure Functions e Brainkin	27
3	Obiettivi e analisi dei requisiti	31
3.1	Contesto iniziale	31
3.2	Obiettivo	32
3.3	Analisi dei requisiti	32
4	Azure Functions e Kubesuite: progettazione e implementazione	35
4.1	Progettazione architetturale	35
4.1.1	Architettura di Kubesuite	36
4.1.2	Pipeline e Dockerfile	38
4.1.3	Approfondimento: ricavare i dati di test dai Pod in esecuzione	38
4.2	Implementazione di Kubesuite	39
4.2.1	Caricare le definizioni degli oggetti dai file YAML	42
4.2.2	Avviare l'ambiente di test	44
4.2.3	Estrarre il log ed eseguire il test	45
4.2.4	Terminare l'ambiente di test	46
4.3	Implementazione della pipeline	47
4.3.1	Primo stage: Build Azure Function as Docker Image	50
4.3.2	Secondo stage: Test Azure Function	53
5	Testing automatico di Azure Function con Kubesuite	55
5.1	Struttura e funzionamento di una Function di test	55
5.2	Kubesuite: esempi d'uso	58
5.2.1	Quartz.NET	58

5.2.2	MQTT	63
6	Conclusioni e sviluppi futuri	71
A	Codici di esempio ed approfondimenti	73
A.1	Leggere lo stato di un Pod	73
A.2	Leggere lo stato di Deployment	74
A.3	Classe di test con Kubesuite	75
A.4	Dockerfile per i test su pipeline con Azure CLI, .NET Core, kubectl e helm	77
A.5	Dockerfile per Azure Functions	78
A.6	Pipeline YAML di compilazione e test per Azure Functions . . .	79
B	Test Quartz.NET	83
B.1	YAML per test di Quartz.NET	83
B.2	Log di test Quartz.NET	84
C	Test MQTT	89
C.1	YAML per test di MQTT	89
C.2	Log di test MQTT	91
	Bibliografia	95

Elenco delle figure

1.1	Classificazione dei <i>jobs</i>	13
1.2	Container vs Macchina virtuale (da [13])	16
1.3	Struttura a <i>layer</i> dei container Docker (immagine tratta da https://docs.docker.com/storage/storagedriver/)	17
1.4	Struttura di una Azure Function App in C#/.NET	23
2.1	Struttura UI di Brainkin	26
2.2	Architettura a layer di Brainkin	28
2.3	Esempio di utilizzo di <i>RuleEngine</i>	30
3.1	Diagramma UML <i>use cases</i> per un sistema di test automatico per Azure Functions su k8s	33
4.1	Diagramma di contesto di Kubesuite	36
4.2	<i>Data Flow Diagram</i> di livello 1 di Kubesuite	37
4.3	Diagramma pipeline CI/CD	39
4.4	Diagramma UML della classe <code>KubernetesCreator</code>	41
4.5	Diagramma UML della classe <code>YamlLoader</code>	44
4.6	Repository estensioni	49
4.7	Contenuto degli <i>artifact</i> pubblicati dal job Build	51
5.1	Struttura del log di una Azure Function App	57
5.2	Diagramma di classi UML per il test su Quartz.NET	59

5.3	Riepilogo esecuzione pipeline per Quartz.NET	61
5.4	Riepilogo esecuzione stage per la pipeline di Quartz.NET	62
5.5	Riepilogo esecuzione job per la pipeline di Quartz.NET	62
5.6	Riepilogo test per Quartz.NET	63
5.7	Funzionamento di MQTT	64
5.8	Diagramma di classi UML per il test su MQTT	67
5.9	Riepilogo esecuzione pipeline per MQTT	69
5.10	Riepilogo esecuzione stages per la pipeline di MQTT	70
5.11	Riepilogo esecuzione job per la pipeline di MQTT	70
5.12	Riepilogo test per MQTT	70

Elenco delle tabelle

1.1	Confronto tra i 4 approcci al DevOps	5
1.2	Ciclo di vita di un Pod [7]	19
1.3	Ciclo di vita di un container [7]	20
1.4	Step di testing per i microservizi	24
5.1	TriggerTime del test di Quartz.NET e differenza tra tempi consecutivi	63
A.1	Possibili Reason di errore per i container in avvio	73

Capitolo 1

Modelli e tecnologie di sviluppo agili per le architetture a microservizi

In questo capitolo verranno introdotte le tecnologie utilizzate nella realizzazione del progetto. Si partirà da una panoramica sulla metodologia DevOps (con un approfondimento sulla piattaforma Azure DevOps) e di come questa venga applicata nell’ambito dello sviluppo di architetture a microservizi. Seguirà poi un’ulteriore sezione dedicata ad alcune tecniche e tecnologie per l’implementazione dei microservizi: containerizzazione (Docker), orchestrazione (Kubernetes) e serverless computing (Azure Functions).

L’attività di ricerca nella letteratura si è concentrata soprattutto sulle “Systematic Literature Review” per le parti più teoriche e per le definizioni. Gli articoli e le fonti prese in considerazione provengono per la quasi totalità dall’archivio digitale della IEEE (<https://ieeexplore.ieee.org/>) e dell’ACM (<https://dl.acm.org/>). Le parole chiave utilizzate per la ricerca sono state: *DevOps*, *Kubernetes*, *Microservice*, *Serverless computing*, *Microservice testing*, *Continuous Integration*, usate anche combinandole tra di loro per raffinare le ricerche.

1.1 DevOps e Continuous Practices (CP)

Definire in maniera rigorosa il concetto di *DevOps* è un compito arduo: trattandosi di un concetto relativamente nuovo (i primi movimenti per il DevOps

sono nati intorno al 2009 [2][16]), all'interno della letteratura ogni autore tende a dare una sua definizione. Leonardo Leite *et al.*, concentrandosi sul lato più metodologico e concettuale della questione, riportano la seguente definizione [16]:

DevOps (Development and Operations) è uno sforzo collaborativo e interdisciplinare all'interno di un'organizzazione per automatizzare il rilascio continuo di nuove versioni del software, garantendo allo stesso tempo correttezza e affidabilità.

L'articolo [1], invece, si concentra sul lato più tecnico e definisce *DevOps* come

Insieme di pratiche con lo scopo di ridurre il tempo tra un cambiamento all'interno di un sistema e il suo trasferimento all'interno dell'ambiente di produzione, mantenendo allo stesso tempo la qualità sia del codice prodotto che del processo di rilascio. Tutte le tecniche che hanno questo scopo vengono chiamate *pratiche DevOps* (*DevOps practices*)

Da un punto di vista più generale, possiamo considerare DevOps come

una branca emergente dell'ingegneria del SW e un approccio metodologico che utilizza dei team cross-funzionali per testare e rilasciare SW velocemente e in maniera affidabile in modo automatico [18], nonché una naturale evoluzione dei metodi agili, i quali non hanno mai messo troppa enfasi sul processo di rilascio e distribuzione del software [16].

Da queste tre definizioni, si possono ricavare alcune considerazioni:

- all'interno delle DevOps practices, è importante garantire la *qualità* del software prodotto
- lo scopo principale di DevOps è il rilascio continuo nel tempo del software (come per i metodi agili).

Per raggiungere questi scopi, il focus principale di DevOps è sulla riduzione del gap tra il team di sviluppo (Dev) e quello di IT (Ops) all'interno delle organizzazioni.

Per ridurre questo "gap" nella letteratura sono stati proposti tre possibili approcci [18]:

- Mix Responsibilities: gli ingegneri hanno la responsabilità sia dello sviluppo che del reparto IT;
- Mix personnel: aumentare la comunicazione tra Dev e Ops (mantenendo però i ruoli separati);
- Bridge team: introdurre un DevOps team separato con il compito di fare da ponte tra Dev e Ops.

1.1.1 I 4 concetti fondamentali di DevOps

All'interno dell'articolo [16] vengono individuati quattro concetti fondamentali di DevOps:

Process

Uno dei principali risultati che DevOps mira a raggiungere è quello di ridurre rischi e costi, aumentare la qualità dei prodotti sw e la soddisfazione del cliente, attraverso release frequenti e affidabili. Per raggiungere questi scopi DevOps promuove l'utilizzo di metodi agili e cicli di feedback molto brevi.

People

Come indicato in precedenza, lo scopo principale di DevOps è quello di "abbattere" la normale distinzione tra team di sviluppo e team di IT. Il problema sta nel come questo debba essere fatto e che cosa voglia dire essere un operatore DevOps. In questo contesto sono state proposte due visioni differenti [18]:

- "DevOps as culture": la conoscenza è condivisa tra sviluppatori e DevOps/IT Ops, per cui i primi sono messi al corrente del funzionamento delle infrastrutture automatizzate (pipelines, server, ecc...), mentre i secondi conoscono il codice e aiutano i primi a risolvere i problemi. Inoltre i DevOps hanno il compito di scrivere le user stories, suddividere lo sviluppo in task e assegnarle agli sviluppatori.
- "DevOps as job descriptions": compito dei DevOps è quello di mettere a disposizione degli sviluppatori le infrastrutture (pipelines, server di deployment, ...).

In entrambi gli approcci la fase di rilascio è automatizzata (attraverso le pipeline) ed è gestita dal team DevOps/IT Ops.

Delivery

La strategia utilizzata da DevOps per raggiungere gli obiettivi illustrati nella sezione *Process* è l'automazione delle *pipeline di deployment*¹. Per fare ciò, uno dei requisiti necessari per adottare DevOps è l'adozione delle *Continuous Practices* (vedi sezione 1.1.3). All'interno di questo concetto vengono inclusi anche gli ambiti di utilizzo in cui DevOps si è rivelato utile e/o efficace nell'aumentare la qualità del prodotto finale. Uno di questi è la creazione e il versionamento di microservizi ed API (vedi sezione 1.2). Il concetto di *Delivery* è quello più universalmente accettato e consolidato.

Runtime

Oltre a rilasciare velocemente le nuove versioni è altrettanto importante che queste siano affidabili e stabili, motivo per cui due delle caratteristiche principali di DevOps sono il monitoraggio costante delle metriche, sia di business (feedback degli utenti) che di più basso livello (codice, infrastruttura, test,...), e quella di poter effettuare esperimenti direttamente in produzione, utilizzando delle opportune tecniche di *Continuous Deployment*.

1.1.2 Implementare DevOps: quattro approcci

Si illustrano ora quattro approcci diversi per l'implementazione di DevOps all'interno di una organizzazione tratti da [18] e riassunti in tabella 1.1.

Developers-Ops

I senior development gestiscono l'infrastruttura automatizzata di CI/CD all'interno di un sistema cloud ibrido². Il team di Ops si occupa di gestire l'infrastruttura fisica presente in sede. Gli altri sviluppatori scrivono il codice,

¹Una *pipeline di deployment* è la descrizione del processo che, preso il codice sorgente, lo distribuisce come prodotto software finito dopo aver eseguito tutte le fasi di test previste.

²Un *cloud ibrido*, a volte definito ibrido cloud, è un ambiente di calcolo che combina un data center locale, definito anche cloud privato o *on-premise*, con un cloud pubblico (gestito da un provider esterno)[20]

Approccio	Infrastruttura	Gestione infrastruttura
Developers-Ops	Ibrida	Senior Developers (cloud), IT Ops team (on-premise)
Developes-Outsourced Ops	Solo cloud	Senior Developers
Developers-DevOps	Solo cloud	DevOps team
DevOps bridge	Ibrida	DevOps Team (cloud), IT Ops team (on-premise)

Tabella 1.1: Confronto tra i 4 approcci al DevOps

lo rilasciano sulle pipelines messe a disposizione e gestiscono autonomamente le applicazioni. In questo caso il senior development è visto come un facilitatore delle pratiche di DevOps.

Developes-Outsourced Ops

Simile al Developers-Ops, solo che l'ambiente di deploy è cloud-based, quindi non c'è la necessità del team di Ops.

Developers-DevOps

Il team di DevOps si occupa di creare e gestire sia le pipelines che l'infrastruttura cloud. Gli sviluppatori si occupano solo di rilasciare il codice

DevOps bridge

Utilizzato in ambienti ibridi. Come nell'approccio Developers-DevOps, gli sviluppatori hanno solo il compito di implementare codice, lasciando la gestione delle pipelines di deploy e delle infrastrutture al team di DevOps. Questi, a loro volta, fanno riferimento al team di IT Ops per quanto riguarda la gestione dell'infrastruttura locale. In questo approccio, ognuno è responsabile delle proprie azioni. Si tratta dell'approccio più utilizzato, anche se molti sono concordi sul fatto che non sia l'approccio giusto per implementare DevOps.

1.1.3 Implementare DevOps: le Continuous Practices

Le tre *DevOps practices* più conosciute e utilizzate (chiamate in questo elaborato *Continuous Practices*, abbreviato CP) sono *Continuous Integration*, *Continuous Delivery*, *Continuous Deployment*, di cui l'articolo [21] riporta le seguenti definizioni:

Continuous Integration (CI)

Pratica di sviluppo nel quale i membri del team integrano i cambiamenti nel codice frequentemente. Include anche il *building*³ e il testing automatico del software. La CI è considerata il primo step di adozione delle *Continuous Practices*.

Continuous DELivery (CDE)

Insieme di pratiche e processi con lo scopo di assicurarsi che un'applicazione sia sempre pronta ad essere rilasciata in produzione dopo aver superato i test e i controlli di qualità.

Continuous Deployment (CD)

Simile al CDE, con la differenza che il CD è un processo *automatico push-based* (il delivery è provocato dai cambiamenti nel codice), mentre il CDE è un processo *manuale pull-based* (avviene in seguito a una richiesta esplicita). Di conseguenza possiamo considerare il CD come un CDE continuato, che, a differenza del CDE, non può essere applicato a tutti i contesti. Esistono infatti alcuni domini applicativi in cui non è accettabile interrompere frequentemente l'esecuzione del software per eseguire aggiornamenti, come ad esempio i sistemi critici⁴. Su questa definizione, tuttavia, è ancora in corso il dibattito accademico e industriale.

³Il *building* è il processo di conversione del codice sorgente in prodotti standalone utilizzabili da un computer, come eseguibili o pacchetti software.

⁴Si definiscono *sistemi critici* tutti quei sistemi in cui l'interruzione del servizio può portare a conseguenze gravi e/o catastrofiche per gli utilizzatori

1.1.4 Le *pipelines*

Lo strumento principale per implementare le CP sono le (*deployment*) *pipelines*. La parte più critica dell'implementazione di una deployment pipeline è l'automazione di questa per due motivi principali:

1. Alcune task manuali non possono essere evitate (ad esempio quelle di quality assurance),
2. Attualmente non esistono degli standard sulle pipelines, anzi ogni piattaforma adotta il suo.

La letteratura [21] divide gli strumenti e i software utilizzati nelle pipelines in sette categorie:

1. Sistemi di controllo versione (Git⁵): in questo stadio gli sviluppatori effettuano il push del codice su repository remoti (es. GitHub⁶);
2. Code management e strumenti di analisi (SonarQube⁷): analizzano e calcolano delle metriche di qualità del codice;
3. Build system (Make⁸, Maven⁹, MSBuild¹⁰): tool automatici per il building del SW;
4. Continuous Integration Server (Jenkins¹¹, Azure DevOps¹², GitHub, Gitlab CI¹³): controllano il repository di codice ed eseguono i tool automatici di building del SW. Alcuni integrano anche gli strumenti di code management e di effettuare il deploy del software senza passare per ulteriori server;
5. Tool di testing (JUnit¹⁴, XUnit¹⁵);

⁵<https://git-scm.com/>

⁶<https://github.com/>

⁷<https://www.sonarqube.org/>

⁸<https://www.gnu.org/software/make/>

⁹<https://maven.apache.org/>

¹⁰<https://docs.microsoft.com/it-it/visualstudio/msbuild/msbuild?view=vs-2019>

¹¹<https://www.jenkins.io/>

¹²<https://azure.microsoft.com/it-it/services/devops/>

¹³<https://about.gitlab.com/>

¹⁴<https://junit.org/junit5/>

¹⁵<https://xunit.net/>

6. Configuration and Provisioning (Puppet¹⁶, Docker¹⁷): sistemi di configurazione degli ambienti di testing e deploy;
7. CD/CDE Server (Jenkins, Azure DevOps): gestiscono il rilascio e il versionamento automatico del software in ambienti diversi.

1.1.5 Ostacoli e pratiche comuni nell'adozione di DevOps e delle CP

Ostacoli architetturali

Molti studi confermano che la maggior parte delle decisioni e dei cambiamenti atti a implementare la CDE sono architetturali[21]. Architetture monolitiche e con un alto livello di accoppiamento tra le componenti sono poco testabili, con la conseguenza che le dipendenze (sia software che tra i team di sviluppo) devono essere così trasferite anche alle pipeline di deployment[16]. Per questi motivi per adottare con successo le CP è necessario utilizzare architetture modulari e flessibili, in modo che ogni singolo componente possa essere testato e rilasciato in modo indipendente dagli altri. In virtù di questa scelta è anche molto importante definire chiaramente e fin da subito le interfacce delle varie componenti. Da questa descrizione risulta come le architetture migliori per il DevOps sono quelle a microservizi (*Microservices architectures o MSA*), che vengono illustrate in modo più approfondito nella sezione 1.2.

Ostacoli organizzativi

Alcuni dei maggiori ostacoli all'adozione della CI all'interno dei team di sviluppo sono di organizzazione del lavoro, in particolare la mancanza di una strategia di test definita (la quale mancanza porta anche a produrre test di bassa qualità) e di suddivisione del lavoro, soprattutto se il team di sviluppo lavora da remoto[21]. A questo si aggiunge il fatto di lavorare con una tecnologia nuova, quindi avendo a disposizione ancora poche persone con le competenze adeguate alla situazione. Le strategie più in uso per risolvere questi problemi sono:

¹⁶<https://puppet.com/>

¹⁷<https://www.docker.com/>

- Adozione del *test-driven development* e delle strategie di branching¹⁸ dei repository;
- Adozione del "*cross-team testing*", ossia il testing di un modulo viene svolto da un team che non è stato coinvolto nella sua implementazione);
- Scomposizione della fase di sviluppo su tre macro-aree[21]:
 1. Interface Module: sviluppo delle interfacce della componente;
 2. Platform Independent Module: sviluppo delle parti indipendenti dalla piattaforma di esecuzione (solitamente le astrazioni dei moduli nativi);
 3. Native Module: sviluppo delle parti dipendenti dalla piattaforma di esecuzione.
- Utilizzo delle *release parziali*:
 - Rilascio limitato solo a certi utenti (*canary deployment*);
 - Nascondere e disabilitare le funzionalità nuove (o problematiche) agli utenti (*dark deployment*);
 - Rolling back immediato alle versioni stabili: in caso di errori o malfunzionamenti di una certa versione di un software o di una configurazione, si procede a reinstallare una versione precedente (e funzionante) del medesimo software o della configurazione in questione;
- Introdurre cambiamenti nella struttura lavorativa, definendo nuove figure professionali specifiche, oppure nuove politiche di organizzazione del lavoro (rotazione dei membri, limiti di tempo, ...).

Ostacoli comunicativi

Nell'adottare DevOps non va sottovalutato l'aspetto comunicativo, sia con i clienti, ma soprattutto all'interno del team si sviluppa, considerato che è anche un obiettivo dichiarato di DevOps. Poca comunicazione porta a poca trasparenza nell'intero processo, poca trasparenza porta a una minore efficacia nell'utilizzo del mezzo, poca efficacia porta scetticismo e poca fiducia nel

¹⁸Il *branching* è una tecnica di sviluppo che consiste nel dividere il repository di codice in rami di sviluppo indipendenti (detti *branch*), i quali vengono successivamente uniti (*merge*) per ottenere il sorgente finale.

mezzo[21]. Sull'aspetto della comunicazione con gli stakeholders, oltre ai già noti problemi di definizione dei requisiti, si aggiungono alcune problematiche relative all'utilizzo della CD, in particolare sul fatto che non tutti i domini sono adatti al rilascio continuo (domini safety critical), ogni dominio ha una sua configurazione specifica (per cui è difficile poi generalizzare il processo) e molti preferiscono avere aggiornamenti meno frequenti in cambio di maggiore stabilità[21]. Alcune delle soluzioni proposte sono:

- Utilizzo massivo dei *changelog*¹⁹ e della documentazione;
- Adozione di uno dei quattro approcci presentati in sezione 1.1.2;
- Coinvolgimento dell'intero team di sviluppo nel processo di deployment, in particolare del cliente:
 - "lead customer": i clienti, oltre a far parte del processo di sviluppo, vengono istruiti alle pratiche e ai concetti della CD
 - "triage meeting": il cliente organizza i meetings con gli sviluppatori e decide i cambiamenti e le priorità dei requisiti.
 - In generale, il coinvolgimento del cliente è importante per l'adozione della CD/CDE, in particolare per quelle organizzazioni che non hanno i mezzi sufficienti per adottare la CD.

1.1.6 Approfondimento: Azure DevOps

Azure DevOps è la piattaforma in-cloud proprietaria di Microsoft per la gestione di progetti di sviluppo software. Su Azure DevOps ad ogni utente è associata una o più *organizzazioni*, la quale è un'insieme di *progetti*. Ogni progetto di Azure DevOps ha a disposizione le proprie istanze di questi cinque servizi:

- Azure Boards: per gestire la pianificazione di milestone e release in contesto di sviluppo agile;
- Azure Repos: servizio di repository Git su cloud. In un progetto possono essere creati più repository git;
- Azure Pipelines: piattaforma di CI/CD;

¹⁹Un *changelog* è la lista dei cambiamenti apportati in una versione di software rispetto alla versione precedente dello stesso

- Azure Test Plans: sistema di gestione e pianificazione dei test (automatici e non) sul software;
- Azure Artifacts: servizio di repository di pacchetti (Nuget, npm, pip) privati.

Azure Pipelines

Azure Pipelines è la piattaforma di Azure DevOps per la creazione e l'esecuzione di *pipelines* di CI/CD. Per Azure DevOps una *pipeline* è “una rappresentazione del processo di automazione che si vuole eseguire per compilare e testare l'applicazione. Il processo di automazione è definito come una raccolta di attività” [11]. Una *pipeline* può essere definita in due modi: attraverso l'interfaccia grafica oppure in un file YAML dedicato all'interno del repository git su cui viene eseguita la pipeline.

Struttura di una pipeline Una *pipeline* è composta da:

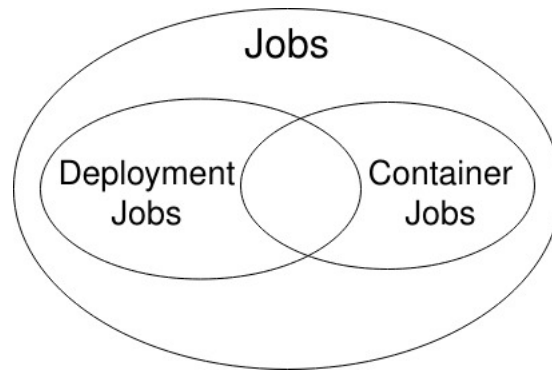
- Un *trigger* che indica quando deve essere eseguita. Un trigger può essere di diversi tipi:
 - CI triggers: la *pipeline* viene eseguita dopo un push sul branch specificato;
 - PR triggers: la *pipeline* viene eseguita all'apertura di una *pull request* sul branch specificato;
 - Scheduled triggers: la *pipeline* viene eseguita a intervalli di tempo pre-impostati
 - Pipeline completion triggers: la *pipeline* viene eseguita in seguito al completamento di una *pipeline* specificata.
 - *none*: la *pipeline* viene eseguita solo in seguito a una invocazione manuale.
- Uno o più *stage*. Uno *stage* si compone di uno o più *jobs* e rappresenta una divisione logica tra i *job* che compongono una pipeline.
- Ogni *job* viene eseguito su un *agent*²⁰ e si compone di uno o più *steps*.

²⁰L'infrastruttura fisica incaricata di eseguire le pipelines. Può essere una macchina virtuale messa a disposizione da Microsoft su cloud oppure una macchina fisica configurata ad-hoc

- Uno *step* è l'unità minima di una pipeline. Può essere una *task* o uno *script* da linea di comando (bash o powershell).
- Una *task* è uno script pacchettizzato e pre-configurato per svolgere delle azioni, come eseguire comandi, invocare delle API, pubblicare o scaricare un *artifact*.
- Un *artifact* (da non confondere con gli Azure Artifacts di sezione 1.1.6) è una raccolta di file pubblicati da una *run*.
- Una *run* è una singola esecuzione di una *pipeline*.
- Un *environment* o *ambiente* è una collezione di risorse in cui viene effettuato un deploy con le credenziali per accedervi (memorizzate da Azure DevOps). Comprende repository remoti, macchine fisiche o virtuali,...
- Ogni *pipeline* ha a disposizione una directory dell'agent nel quale memorizza i repository clonati, gli *artifacts* pubblicati e/o scaricati e tutti i file necessari al suo funzionamento (librerie, file di cache, ...).

Esistono poi due categorie di job (non esclusive, vedi figura 1.1) utilizzate in contesti specifici:

1. I *Deployment Job* servono a racchiudere in un unico job gli step necessari a effettuare un deploy in un dato ambiente. A differenza dai job tradizionali, i deployment job hanno queste caratteristiche:
 - non effettuano in automatico il checkout (download) del repository;
 - effettuano in automatico il download di tutti gli *artifacts* pubblicati nei jobs precedenti;
 - mantengono in memoria la cronologia dei precedenti rilasci;
 - Permettono di definire la strategia di deployment:
 - *runOnce*: gli step vengono eseguiti una volta sola;
 - *rolling*: disponibile solo per le VM. Ad ogni iterazione rilascia solo un piccolo numero di macchine virtuali.
 - *canary*: Simile al rolling, solo che il numero di rilasci cresce ad ogni iterazione ed è disponibile anche per i cluster Kubernetes.
- Un *deployment job* ha un ciclo di vita formato da:
 - *preDeploy*: steps da eseguire prima di effettuare il deploy. Viene eseguito una volta sola a prescindere dalla strategia scelta;

Figura 1.1: Classificazione dei *jobs*

- *deploy*: steps per effettuare il deploy nell'ambiente;
 - *routeTraffic*: steps per spostare il traffico verso la nuova versione rilasciata;
 - *postRouteTraffic*: steps da eseguire in seguito al *routeTraffic*. Solitamente serve a monitorare il traffico verso la nuova versione.
 - *on: Success* e *on: Failure*: steps da eseguire in caso di successo o fallimento delle fasi precedenti. Solitamente si utilizza *on: Failure* per effettuare il rollback.
2. I *Container Job* sono job eseguiti all'interno di container Docker²¹, invece che essere eseguiti direttamente nell'agent. Questo permette un maggiore isolamento sia dei processi che delle dipendenze necessarie ad eseguire un job. La directory della pipeline viene montata all'interno del container come un volume.

1.2 Architetture a microservizi e DevOps

Si è già discusso dell'importanza delle scelte architetturali nell'adozione della filosofia DevOps. Uno degli ambiti applicativi particolarmente adatti all'adozione delle CP e del DevOps è l'implementazione di architetture a microservizi, abbreviati *MSA* (dall'inglese *Microservices Architecture*)²². Più questi sono indipendenti tra loro e presentano un basso livello di accoppiamento con altri

²¹Per le informazioni su Docker e container si rimanda a sezione 1.2.1.

²²Si definisce *architettura a microservizi* un pattern architetturale nel quale le applicazioni sono composte da un insieme di componenti modulari indipendenti tra di loro, detti micro-

microservizi, più questi sono testabili ed è facile effettuare rapidamente il deploy, esattamente i requisiti ricercati per la CI/CD. Altro punto a favore delle MSA è quello di adattarsi efficacemente e rapidamente in contesti fortemente regolamentati [16] o con requisiti in costante evoluzione: è sufficiente infatti ridefinire (o implementare ex-novo) i microservizi necessari allo scopo, senza dover intervenire sull'intero sistema.

Tuttavia, le MSA presentano alcune problematiche:

- eterogeneità dei microservizi, soprattutto quando riguardano aspetti non-funzionali (logging, startup, configurazioni,...),
- le versioni rilasciate devono essere le stesse utilizzate in fase di integration testing, pena il rischio che queste non funzionino correttamente per incompatibilità.

A questo proposito la letteratura ha proposto alcuni pattern per risolvere questi problemi:

- standardizzare un piccolo set fondamentale di microservizi all'interno dell'organizzazione[16],
- utilizzare una pipeline diversa per ogni microservizio[16],
- utilizzare *log aggregator*²³ e sistemi di indicizzazione e ricerca dei microservizi[16],
- isolare codice sorgente, configurazioni e variabili d'ambiente dall'esterno[1],
- utilizzare sistemi automatici per la scalabilità (load balancers, contratti a consumo, ...)[16]
- utilizzare sistemi di compatibilità[16],
- utilizzare sistemi di versionamento della API. Quest'ultimo punto però è controverso, perchè aumenta la difficoltà di deployment e di gestione delle API[1].

servizi. Ogni microservizio comunica verso l'esterno o con gli altri microservizi attraverso delle interfacce ben definite dette API (Application Programming Interface). Ogni microservizio è incaricato di fornire una specifica funzionalità (stabilita a priori) per il funzionamento globale dell'applicazione

²³I *log aggregator* sono servizi che si occupano di intercettare tutto o una parte dell'output di uno o più programmi (errori, informazioni di debug, risultati, ...) e raggrupparlo in un unico luogo per poter essere letto dall'esterno

In questo elaborato si discuterà in particolare di tre tecnologie utilizzate per implementare i microservizi: *containerizzazione*, *orchestrazione* e *serverless computing*.

1.2.1 Containerizzazione: Docker

Un *container* è un'unità di software che impacchetta il codice e tutte le sue dipendenze in modo che l'applicazione venga eseguita in modo rapido e affidabile da un ambiente di elaborazione all'altro[13].

I container sono molto simili alle macchine virtuali: entrambi hanno il proprio filesystem e il proprio ambiente di esecuzione (librerie, variabili d'ambiente, pacchetti installati, ...) isolato rispetto al sistema host, la differenza sta nel fatto che le macchine virtuali emulano sia l'hardware che l'intero sistema operativo, mentre i container emulano solo una parte del sistema operativo, condividendo il kernel, che è lo stesso del sistema host, risultando quindi molto più leggeri in termini di risorse hardware occupate. La differenza tra i due approcci è illustrata in figura 1.2.

Per implementare i container vengono utilizzate due primitive del kernel Linux:

- i Kernel Namespaces: a ogni namespace viene fatto corrispondere un diverso albero dei processi. In questo modo processi su alberi diversi non sono in grado di comunicare tra loro;
- i *cgroups*: impediscono alle applicazioni di acquisire tutte le risorse del computer senza mai rilasciarle;

Un container ha due modi di comunicare con l'esterno:

- attraverso il *port mapping*, per cui una porta TCP/UDP del container viene mappata su una porta TCP/UDP libera del sistema host. Ad esempio: con un mapping 80:8080 si collega la porta 80 del container alla porta 8080 del sistema host. I client esterni per comunicare con il container devono collegarsi alla porta 8080 dell'host,
- montando all'interno del container una o più directory del sistema host. Una directory dell'host montata all'interno di un container si chiama *volume*. In questo modo se l'app va a modificare dei file all'interno di questi percorsi, i cambiamenti si riflettono anche sull'host e viceversa.

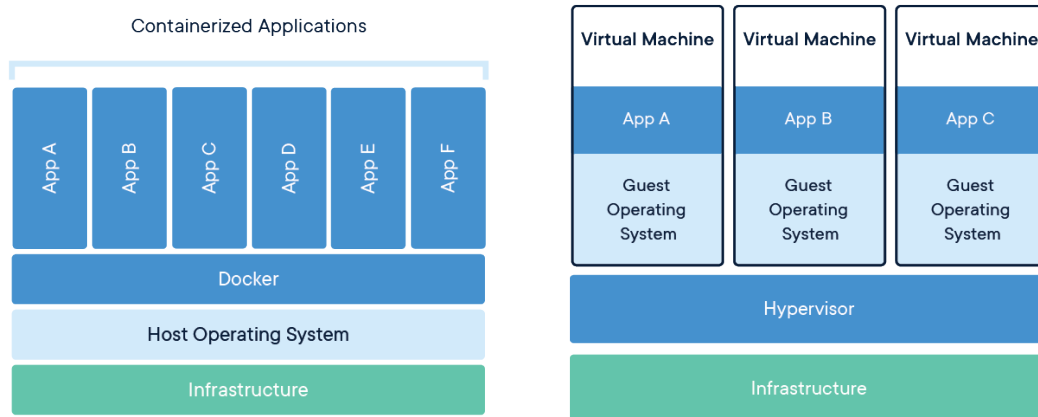


Figura 1.2: Container vs Macchina virtuale (da [13])

Il software più utilizzato per eseguire container è *Docker*. I container vengono istanziati a partire da una *immagine container Docker*. Un'*immagine container Docker* è un pacchetto software leggero, autonomo ed eseguibile che include tutto il necessario per eseguire un'applicazione: codice, runtime, strumenti di sistema, librerie di sistema e impostazioni[13]. Un'immagine container Docker non è altro che un archivio compresso di file e presenta una struttura a layer per descrivere il suo file system. Una volta creata una immagine Docker questa non è più modificabile.

A livello più basso di un container c'è un file system di base e ad ogni layer vengono memorizzate le modifiche al file system rispetto al livello precedente. Questi layer vengono poi uniti tramite UnionFS. UnionFS è un file system di Linux in grado di sovrapporre file e directory separati in modo da creare un unico file system (virtuale) coerente.

Quando un container viene avviato, Docker aggiunge al container un ulteriore layer scrivibile, in cui vengono memorizzate tutte le modifiche al file system del container. Quando il container viene terminato, viene eliminato anche questo layer e quindi vengono eliminate tutte le modifiche effettuate dal container nel suo filesystem, a meno che queste non siano state salvate in un volume.

Nell'immagine 1.3 è mostrato questo schema: a partire da una immagine di base (ubuntu:15.04) vengono istanziati quattro layer immutabili (ognuno contraddistinto da un *hash* di controllo) per costruire l'immagine Docker. Da questa immagine istanzio un certo numero di container Docker, ognuno con il

suo layer scrivibile.

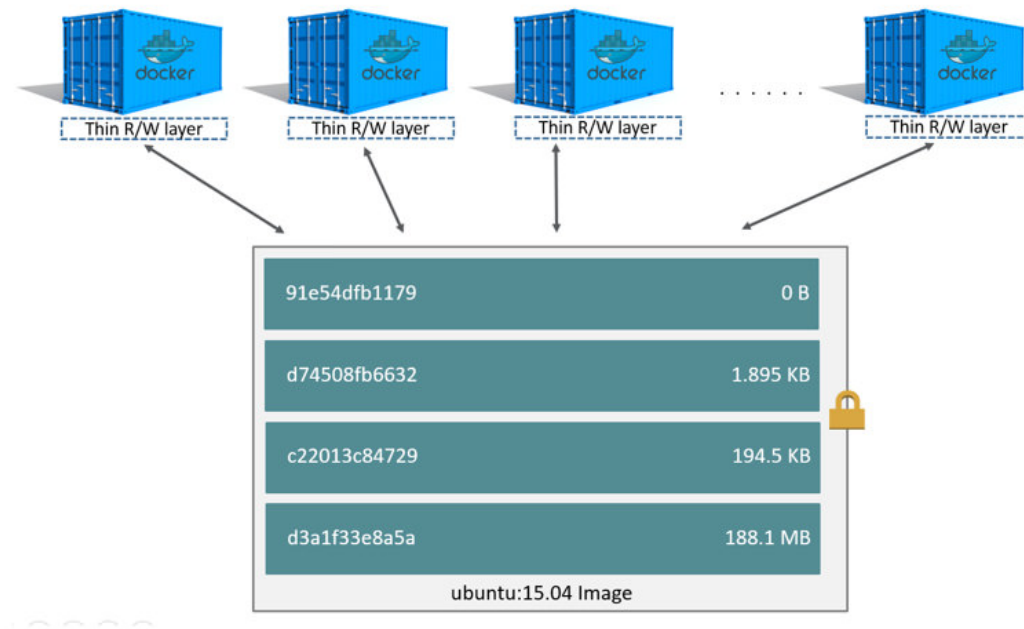


Figura 1.3: Struttura a *layer* dei container Docker (immagine tratta da <https://docs.docker.com/storage/storagedriver/>)

Ogni container ha a disposizione un servizio, denominato *logging driver*. Normalmente il runtime di Docker cattura il contenuto dello standard output e dello standard error del container. Il logging driver indica a Docker come formattare e dove inviare il log catturato (file JSON, journalctl, file di testo, ecc...).

Un immagine Docker viene descritta da un apposito *Dockerfile*. La prima riga del Dockerfile descrive l'immagine di partenza su cui costruire il container, le righe successive descrivono le operazioni da svolgere su questo container (esecuzione di comandi, copia di file dall'host al container, definizione di volumi, ecc...). Ogni riga del Dockerfile rappresenta un layer del container.

Docker esegue immagini presenti in locale o in un *container registry* remoto pubblico o privato. Il *container registry* più conosciuto ed utilizzato è Docker Hub²⁴, anche se tutti i cloud provider più famosi (AWS²⁵, Microsoft

²⁴<https://hub.docker.com/>

²⁵https://aws.amazon.com/it/ecr/?nc2=h_q1_prod_ct_ec2reg

Azure²⁶, Google Cloud²⁷) mettono a disposizione la possibilità di creare dei propri container registry privati.

1.2.2 Orchestrazione di container: Kubernetes

Passo immediatamente successivo alla containerizzazione è l'*orchestrazione* dei container. Definiamo orchestrazione dei container come

l'automazione dei processi di deployment, gestione, scalabilità e networking dei container.[12]

Il software più utilizzato a questo scopo è *Kubernetes*²⁸, conosciuto anche come *k8s* per brevità.

Un cluster Kubernetes è un'insieme di macchine, chiamate nodi, che eseguono container gestiti da Kubernetes. Un cluster ha almeno un Worker Node[5].

I Worker Node ospitano i Pod che eseguono i workload dell'utente. I Control Plane Node gestiscono i Worker Node e tutto quanto accade all'interno del cluster[5].

Tutti i nodi di un cluster k8s comunicano con il Control Plane attraverso le *Kubernetes API*, che vengono esposte agli amministratori attraverso due strumenti: le librerie per i vari linguaggi di programmazione e la riga di comando *kubectl*. Una chiamata alle API non è altro che una richiesta HTTP/S di tipo REST effettuata verso il control plane.

Siccome impostare un cluster Kubernetes è piuttosto complesso, molti cloud provider mettono a disposizione Kubernetes come Paas (Platform as a service), ossia il provider mette a disposizione il cluster k8s pronto all'uso, astraendo le risorse (macchine virtuali, storage, networking, ...) agli utilizzatori.

Per accedere a un cluster k8s è necessario possedere un certificato di autenticazione (non esiste l'accesso con username e password) rilasciato dal control plane e solitamente salvato all'interno di un file all'interno del computer da cui ci si vuole collegare (di default sui sistemi GNU/Linux le credenziali vengono salvate all'interno di `$HOME/.kube/config`).

²⁶<https://azure.microsoft.com/en-us/services/container-registry/>

²⁷<https://cloud.google.com/container-registry>

²⁸<https://kubernetes.io/>

1.2.3 Panoramica delle API di Kubernetes

L'API server è il cuore del control plane di un cluster k8s. Attraverso le API è possibile manipolare e ottenere informazioni sugli oggetti di Kubernetes. Caratteristica fondamentale di Kubernetes è il suo approccio dichiarativo, cioè il fatto che l'utente-amministratore ha il solo compito di descrivere lo stato desiderato dei vari oggetti sul cluster attraverso la riga di comando oppure scrivendolo in file YAML, sarà poi compito del control plane verificare che lo stato sul cluster sia sempre quello desiderato e, in caso contrario, agire di conseguenza.

In seguito sono approfonditi alcuni oggetti di Kubernetes (per brevità *oggetti k8s*, per distinguerli dagli oggetti dei linguaggi di programmazione) che saranno poi utilizzati nel proseguimento dell'elaborato.

Pod

Il *Pod* è l'unità minima di Kubernetes. Un Pod è formato da uno o più container in esecuzione [8], tuttavia è buona norma mantenere un solo container per Pod. Lo stato attuale di un Pod è descritto all'interno del campo **Status**. Questo campo rappresenta lo stato sia del Pod all'interno del suo ciclo di vita (vedi tabella 1.2), sia dei container che lo compongono (tabella 1.3).

Fase	Descrizione
Pending	Il Pod è in fase di creazione
Running	Tutti i container del Pod sono in esecuzione
Succeeded	Tutti i container del Pod sono stati terminati correttamente
Failed	Almeno un container del Pod è terminato con un errore
Unknown	Non è possibile comunicare con il Pod

Tabella 1.2: Ciclo di vita di un Pod [7]

Deployment

Un *Deployment* è un oggetto utilizzato per dichiarare lo stato desiderato di un insieme di pod che eseguono le stesse immagini container Docker, chiamate *repliche* [4]. In un Deployment vengono inserite informazioni quali il numero di repliche desiderate in un certo momento (eventualmente scalabili automaticamente in base al carico di lavoro richiesto). Così come per i Pod, anche i

Fase	Descrizione
Waiting	Il container è in fase di creazione. Possiede un campo Reason che descrive perchè il container è in quello stato
Running	Il container è in esecuzione
Terminated	Il container è terminato. Possiede un campo Reason che descrive perchè il container è terminato

Tabella 1.3: Ciclo di vita di un container [7]

Deployment possiedono uno stato, in questo caso memorizzato come una coda di eventi. Lo stato attuale si ricava dall'ultimo evento e si analizza attraverso tre campi: **Type**, **Status** e **Reason**. I Deployment hanno un tempo massimo entro cui devono essere pienamente operativi. Se ciò non avviene viene dichiarato come fallito e, quindi, non utilizzabile.

Service

Un *Service* è un astrazione per un insieme di pod, che vengono così visti dall'esterno come un unico oggetto di rete. Il control plane ha un proprio server DNS interno al cluster e ogni pod ha un proprio indirizzo IP interno al cluster [10]. Il Service non fa altro che assegnare un nome DNS comune e a bilanciare il carico di rete tra i pod che rispondono allo stesso servizio. In questo modo gli altri pod possono collegarsi a uno qualsiasi dei pod del service usando direttamente il suo nome. I Service solitamente vengono usati in combinazione con i Deployment.

ConfigMap

Una *ConfigMap* è una struttura dati per memorizzare dati non confidenziali in coppie chiavi-valore [3]. I pod possono usare le ConfigMaps per impostare variabili d'ambiente, argomenti per la linea di comando, oppure possono essere create a partire da un file di testo e montate in un pod come un volume.

Secret

Un *Secret* è molto simile alla ConfigMap, con la differenza che serve a memorizzare dati confidenziali come password, token OAuth, certificati tls o chiavi ssh [9]. Il tipo di Secret più comune è *Opaque*, ossia un insieme di coppie

chiave-valore di dati arbitrari dove, a differenza delle ConfigMaps, i valori sono memorizzati in formato `base64`.

Namespace

I *namespaces* sono delle partizioni virtuali di un cluster Kubernetes [6]. Ogni oggetto è associato ad un namespace. In un namespace non è possibile avere due oggetti dello stesso tipo con lo stesso nome, mentre ciò è possibile tra namespace diversi. Quando viene creato un Service, il nome DNS interno assegnatogli è nella forma `nomeService.nomeNamespace.svc.cluster.local`. Per comunicare dall'esterno di un namespace con un Service è necessario usare il nome DNS completo.

1.2.4 Serverless computing

Negli ultimi anni in ambito cloud si è diffuso il paradigma del *Serverless computing*, un servizio in cloud in cui la logica delle applicazioni viene divisa in funzioni che vengono eseguite in risposta a degli eventi (interni o esterni)[19].

A differenza delle classiche applicazioni cloud, in cui le risorse vengono istanziate a priori, nelle applicazioni serverless le risorse necessarie all'esecuzione della funzione vengono allocate solo quando queste vengono invocate dal trigger.

Le piattaforme più famose di serverless computing sono AWS Lambda di Amazon e Azure Functions di Microsoft. Entrambe supportano diversi linguaggi di programmazione ed entrambe pubblicano le immagini docker con il runtime di esecuzione delle funzioni. In questo modo gli sviluppatori non sono costretti a utilizzare il cloud, ma possono distribuire le applicazioni direttamente su Docker o Kubernetes.

Struttura di una Azure Function

Specificando la definizione utilizzata nella sezione precedente, una Azure Function è sostanzialmente una procedura stateless standalone scritta in un certo linguaggio di programmazione (sono supportati C#, Java, Javascript, Typescript, Python, Powershell, Go e Rust) a cui viene associato un *trigger*, uno o più *input bindings*, uno o più *output bindings*. I bindings non sono obbligatori.

Un trigger è colui che causa l'esecuzione di una Function. A un trigger sono associati dei dati che possono essere utilizzati all'interno della Function.

Un input binding è un dato ricevuto in ingresso dalla Function, di solito provenienti da database o sistemi di storage.

Un output binding è un simile a un parametro passato per risultato: la Function utilizza gli output binding per ritornare valori verso sistemi di storage o database.

Più Functions possono essere racchiuse all'interno della stessa *Function App* (se si prevede di rilasciare le Function in cloud), oppure all'interno dello stesso container (se si rilascia su Docker/k8s).

Gli eseguibili delle Functions in una Function App si trovano in una directory radice contenente il file `host.json`. In questo file è riportata la configurazione per il runtime. La directory `bin/` contiene gli eseguibili e le librerie per il runtime. Per le singole Functions la struttura delle directory dipende dal linguaggio utilizzato. Nel caso di C# compilato l'eseguibile vero e proprio si trova nella directory `bin/`, mentre nella directory radice c'è una directory per Function. All'interno di queste si trova il file `function.json` in cui sono salvate le informazioni sulla Function come il trigger, i bindings e la posizione dell'eseguibile. Questo file viene generato automaticamente a partire dai metadati della Function, espressi attraverso gli attributi di C#.

In figura 1.4 è visibile la struttura di una Function App. Nel rettangolo centrale è illustrata la struttura del file system.

Estensioni di Azure Functions

Il runtime di Azure Functions integra al suo interno vari tipi di trigger e bindings (che d'ora in avanti chiamati anche *estensioni*), tuttavia è possibile svilupparne di nuovi secondo le esigenze. Una estensione, infatti, non è altro che un pacchetto Nuget che viene installato all'interno del runtime.

Dal punto di vista dell'architettura triggers e bindings sono molto simili. Entrambi sono definiti a partire da una serie di campi e delle regole per raccogliere dati dalle sorgenti e assegnare questi dati agli attributi (chiamate `BindingRules`). Per fare ciò, implementano una serie di interfacce.

La differenza tra i due sta nelle interfacce implementate:

- un trigger implementa l'interfaccia per definire un *listener*, ossia un processo in esecuzione in background che rimane in attesa di un certo evento

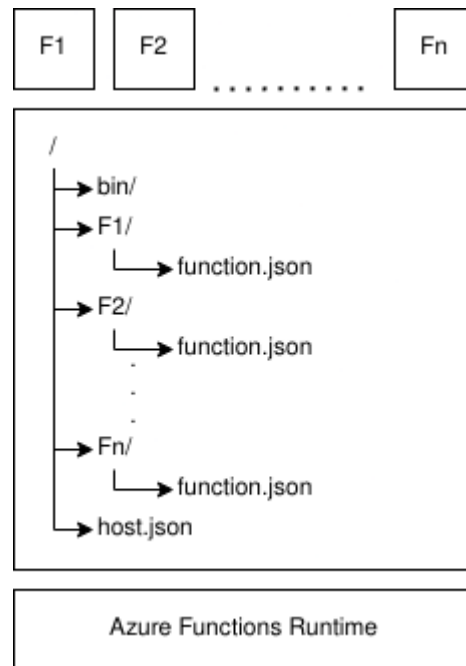


Figura 1.4: Struttura di una Azure Function App in C#/.NET

per poter processare i dati da esso e invocare la Function assegnatagli,

- un binding implementa l'interfaccia per definire un *collector*, ossia un middleware che collega la Function alla sorgente di dati richiesta.

1.3 Testing di microservizi e framework automatici di test

L'automazione della fase di testing è un requisito fondamentale per l'adozione della CI in un'organizzazione. L'articolo [17] elenca sei livelli di testing per i microservizi riassunti in tabella 1.4.

Nel caso di microservizi sviluppati tramite Azure Functions, nel secondo step vengono testate le singole Functions, nel terzo si testa l'intera Function App nel complesso, anche se spesso queste due fasi vengono accorpate in una.

Testare un architettura a microservizi, tuttavia, presenta alcune problematiche, legate soprattutto a tre fattori[22]: l'aumento della complessità del sistema all'aumentare dei microservizi installati (performance, comunicazione

Step	Descrizione
Unit test	Test di singole funzioni e/o oggetti
Integration test	Test di funzionamento tra più unità di un microservizio. Solitamente in questa fase si testano le interfacce verso servizi e moduli esterni al sistema
Single microservice test	Il funzionamento di singolo microservizio viene testato nella sua interezza
Contract test	A ogni servizio è associato un contratto. In questa fase si verifica che, nel comunicare tra loro, tutti i servizi rispettino i contratti
E2E (End-to-end) test	Test di funzionamento globale del sistema dal punto di vista dell'utente
Cloud test	In questa fase vengono eseguiti test sul sistema (locale o già in cloud) utilizzando diverse configurazioni messe a disposizione dal cloud provider.

Tabella 1.4: Step di testing per i microservizi

tra i microservizi, ...), l'automazione delle fasi di test e l'ottenimento di feedback da parte del framework di test (attualmente non esistono framework di test in grado di dare feedback affidabili in applicazioni basate su filesystem, database o protocolli di rete).

Nel proseguimento dell'elaborato verrà proposto un approccio al testing di integrazione di microservizi su Kubernetes.

Capitolo 2

Sistemi IoT per il monitoraggio di una linea di produzione: il prodotto Brainkin

In questo capitolo verrà presentato un esempio di architettura a microservizi nell'ambito di un prodotto commerciale, nello specifico *Brainkin*.

Dopo un breve panoramica dell'interfaccia e degli aspetti funzionali, verrà svolta una breve analisi sull'architettura del prodotto, con particolare enfasi sul ruolo delle Azure Functions nel funzionamento del prodotto, che sarà un tema centrale nei capitoli successivi.

2.1 Interfaccia e funzionalità di Brainkin

Brainkin è una piattaforma di *IoT industriale*¹ per il monitoraggio delle linee di produzione, sviluppata internamente da Beantech.

Brainkin si rivolge alle aziende manifatturiere e ai produttori di macchinari per l'industria manifatturiera. Questi ultimi, infatti, acquistano il prodotto per preinstallarlo nelle macchine, diventando quindi un efficace veicolo di diffusione del prodotto stesso.

L'obiettivo principale di Brainkin è il cosiddetto *Asset Monitoring*, ossia avere a disposizione in ogni momento la situazione dell'intera produzione nei

¹Per *IoT industriale* si intende quella branca dell'Internet of Things che si occupa della sua applicazione in ambito industriale

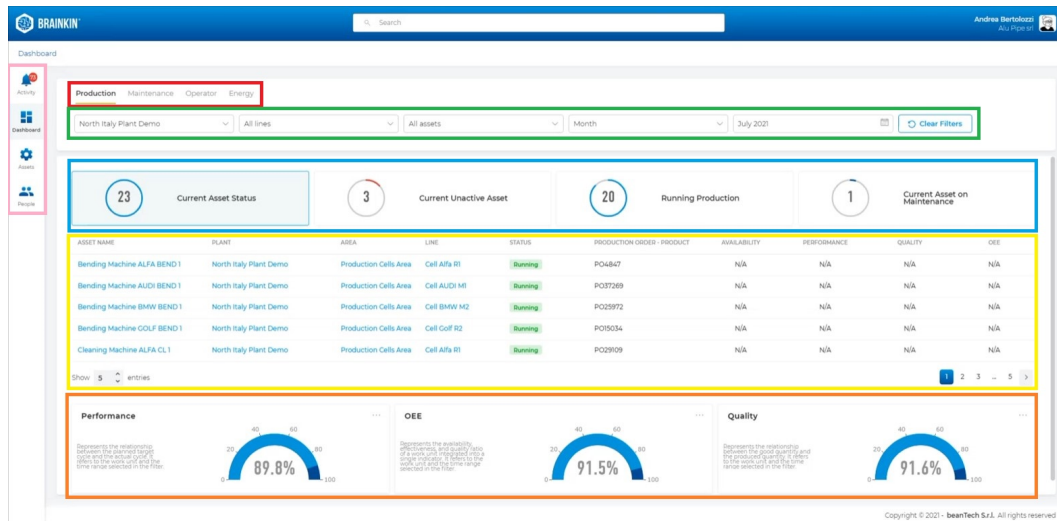


Figura 2.1: Struttura UI di Brainkin

singoli *asset*. Gli asset sono di tre tipi: macchine, stazioni di controllo qualità e magazzini.

La struttura della UI è praticamente identica per tutte le sezioni dell'interfaccia ed è mostrata in figura 2.1.

La barra laterale (cerchiata in rosa) serve a selezionare la sezione dell'interfaccia. Le quattro sezioni sono:

- **Activity:** qui sono presenti le notifiche. Le notifiche mostrate all'utente variano in base al suo ruolo.
- **Dashboard:** mostra il riepilogo globale di tutte le informazioni raccolte da Brainkin.
- **Asset:** qui i singoli asset vengono divisi per impianto di produzione. Selezionando poi uno specifico asset viene mostrato nel dettaglio.
- **People:** questa sezione serve per gestire gli operatori di manutenzione. Il responsabile assegna una manutenzione a un operatore, questo riceve una notifica sulla propria applicazione. Durante una manutenzione l'interfaccia mostra all'operatore le informazioni sull'oggetto della manutenzione e una checklist delle operazioni da svolgere. L'operatore deve aprire e chiudere la manutenzione dall'interfaccia di Brainkin.

Scorrendo il resto dell'interfaccia dall'alto verso il basso sono presenti:

- le tab dell'interfaccia (cerchiate in rosso). Queste vengono mostrate o nascoste in base all'utente connesso e alla sezione scelta. Tra le tab più significative ci sono:
 - Information: contiene un riepilogo sull'elemento visualizzato. Se prevista, è sempre presente a prescindere dal ruolo dell'utente,
 - Production: contiene le informazioni sulla produzione (pezzi prodotti, efficienza, ecc. . .) e lo storico delle manutenzioni,
 - Maintenance: contiene le informazioni sulla manutenzione (calendario delle manutenzioni, documentazione, fermi macchina),
 - Energy: contiene le informazioni relative ai consumi energetici;
- i filtri temporali (cerchiati in verde). I dati delle macchine vengono raccolti regolarmente e non esiste una visualizzazione in tempo reale dei dati. Lo scopo di Brainkin non è fornire all'utente un monitoraggio in tempo reale, ma solo raccogliere e analizzare i dati provenienti dalle macchine per realizzarne uno storico;
- delle informazioni di riepilogo (cerchiate in blu) sottoforma di *card* cliccabile. A seconda della card selezionata, cambia il contenuto della sezione sottostante (cerchiata in giallo).
- i dati veri e propri sono raccolti e mostrati all'utente prevalentemente in forma tabellare,
- infine dei grafici di riepilogo (cerchiati in arancione) scaricabili dalla UI come grafici Excel.

2.2 Azure Functions e Brainkin

Brainkin si basa su una architettura a microservizi. Questi microservizi vengono rilasciati su un cluster Kubernetes in cloud oppure on-premise. Viene seguito il seguente flusso di dati:

1. i macchinari scrivono i dati sui loro registri interni;
2. i macchinari sono collegati a dei broker di protocolli IoT. I principali sono MQTT, RabbitMQ e OPC UA. Le macchine inviano direttamente

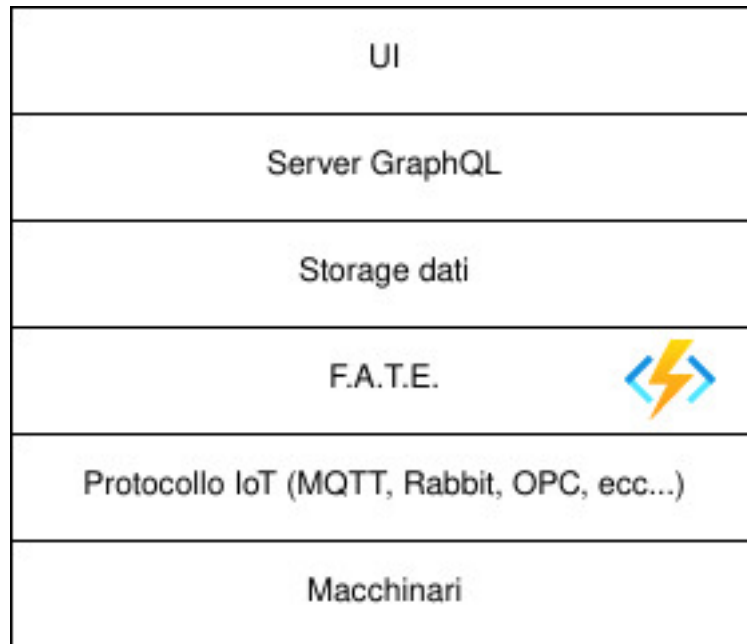


Figura 2.2: Architettura a layer di Brainkin

i dati ai broker, oppure questi vengono letti dall'esterno (come nel caso di OPC UA);

3. il backend di Brainkin elabora questi dati sottoforma di stringhe JSON e le invia verso sistemi di storage, code di messaggi o database;
4. dei microservizi GraphQL interrogano i database. I servizi GraphQL sono una variante dei classici servizi REST;
5. la UI di Brainkin si interfaccia a GraphQL, ai file e alle code per mostrare l'interfaccia di Brainkin all'utente.

Sulla base di questo flusso Brainkin può essere rappresentato utilizzando un'architettura a strati, come illustrato in figura 2.2.

Le Azure Functions vengono utilizzate nel terzo livello. L'acronimo *F.A.T.E.* utilizzato in figura 2.2 sta per *Filter, Aggregate, Transform, Enrich* e riassume perfettamente le funzionalità di questo livello: filtrare, aggregare, trasformare e arricchire con dei metadati i dati provenienti dai protocolli IoT al secondo livello. Per ottenere questi dati si utilizzano due metodi:

- tramite un TimerTrigger la Function legge periodicamente i dati,

- tramite un trigger apposito che invoca la Function.

In entrambi i casi si rende necessario avere a disposizione una ricca libreria (chiamata **Beantech.SF**) di estensioni sviluppate appositamente per supportare tutti i protocolli e le possibili destinazioni dei dati elaborati. Inoltre è possibile che i dati passino attraverso più Functions di elaborazione prima di essere salvati, motivo per cui quasi tutti i protocolli oltre ad essere implementati come trigger, sono implementati anche come bindings.

L'operazione di *F.A.T.E.* viene svolta dal *RuleEngine* di Beantech. RuleEngine è una libreria interna a Beantech altamente configurabile che elabora dei dati contenuti in delle stringhe JSON, fornendo in output il risultato, anch'esso in formato JSON. La caratteristica principale di RuleEngine è che permette di definire delle elaborazioni complesse sui dati in input attraverso delle pipeline scritte in formato JSON, che ne descrivono i passaggi.

In questo modo le Azure Functions che compongono il backend di Brainkin sono altamente standardizzate, infatti il codice della funzione è il medesimo per tutte (esegue semplicemente il RuleEngine). Ciò che le distingue sono i trigger e i bindings che vengono utilizzati e la pipeline JSON di elaborazione.

Si è detto in precedenza che RuleEngine riceve in input e restituisce in output stringhe JSON, mentre non è detto che i vari protocolli utilizzino lo stesso formato. Per risolvere questo problema le estensioni di **Beantech.SF** implementano delle BindingRule che formattano i dati in JSON prima di passarli alla Function.

In figura 2.3 è illustrato un esempio di funzionamento di *RuleEngine*:

1. viene ricevuto in input una stringa JSON da un PLC che conta i pezzi che entrano in produzione,
2. al primo passaggio della pipeline è un filtro: si procede al passo successivo solo se il valore del token `Counter.Value` è diverso da `null`,
3. nel secondo passaggio viene creato un nuovo token JSON (chiamato **Ts**) contenente il Timestamp attuale,
4. nel terzo e ultimo passaggio compongo il JSON in output utilizzando i token `Counter.Value` e `Ts`.

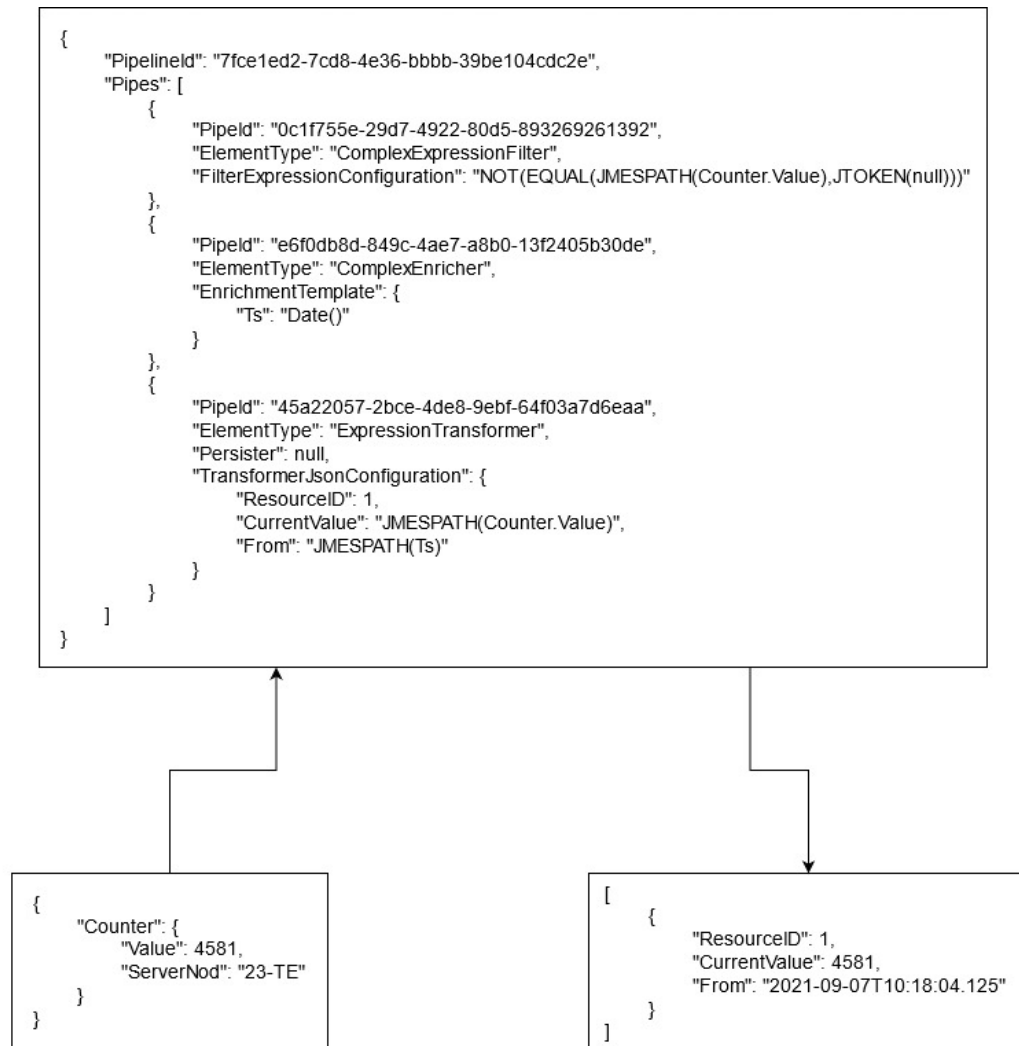


Figura 2.3: Esempio di utilizzo di *RuleEngine*

Capitolo 3

Obiettivi e analisi dei requisiti

In questo capitolo verranno definiti gli obiettivi e verrà svolta l'analisi dei requisiti (funzionali e non) su cui verrà poi sviluppato il progetto Kubesuite.

3.1 Contesto iniziale

Il team di sviluppo di Brainkin utilizza un metodologia di sviluppo di tipo incrementale. Uno degli obiettivi a breve termine del team è quello di adottare l'approccio DevOps, in particolare nello sviluppo delle Azure Functions per Brainkin.

Particolare enfasi all'interno del team è stata data nello standardizzare il più possibile il processo di rilascio dei container contenenti le Azure Functions (obiettivo già parzialmente raggiunto con lo sviluppo della libreria RuleEngine) attraverso l'uso delle pipeline di Azure DevOps. In questo modo il lavoro del team di sviluppo si riduce a scrivere le pipeline JSON per il RuleEngine e sviluppare nuove estensioni.

Le criticità maggiori si presentavano nella fase di testing delle estensioni, la quale si limitava allo scrivere delle Azure Function di test da eseguire in locale e osservarne poi il funzionamento.

Un'altra problematica degna di nota era nell'utilizzo da parte del team di sviluppo di un modello di DockerFile per le Function non adatto, che portava a potenziali problemi di sicurezza, in quanto prevedeva di esporre delle credenziali utente.

3.2 Obiettivo

Alla luce di questa analisi, l'obiettivo prefissato è quello di migliorare il contesto produttivo attuale sia nell'ambito dello sviluppo di Azure Functions sia dei relativi test in modo da fornire al team di sviluppo di Brainkin uno o più strumenti di supporto al testing e al rilascio automatico delle Function da pipeline, in particolare per quanto riguarda le fasi di test successive ai test di unità (vedi tabella 1.4), in modo da superare (o per lo meno limitare) le problematiche presentate.

In quest'ottica il testing delle estensioni si riduce a effettuare i test di funzionamento su delle Azure Functions create appositamente allo scopo.

3.3 Analisi dei requisiti

I requisiti sono stati raccolti in fase di analisi del processo di sviluppo e di rilascio del software adottato dal team di Brainkin. Da questa analisi sono stati estratti alcuni requisiti preliminari (non funzionali):

- ogni test deve essere eseguito in un ambiente creato ex-novo, per cui non devono essere presenti elementi istanziati da test eseguiti in precedenza,
- per lo stesso motivo di cui al punto precedente, una volta concluso il test, l'ambiente di test deve essere riportato allo stato precedente l'esecuzione,
- considerato che i test devono essere eseguiti da una pipeline, l'infrastruttura di test deve essere sufficientemente robusta da non dover mai richiedere l'intervento di operatori umani, anche in caso di errori o mal-funzionamenti,
- per ragioni di coerenza tra l'ambiente di test e quello di produzione di Brainkin, si è deciso che le Azure Function di test dovessero essere containerizzate e mandate in esecuzione su un cluster Kubernetes.

Per ricavare i requisiti funzionali è stata svolta una ulteriore analisi con l'obiettivo di individuare gli stakeholders coinvolti nell'intero processo di sviluppo. Ne sono stati individuati quattro, riassunti nel diagramma UML *use cases* in figura 3.1:

- Sviluppatori: hanno il compito di implementare le estensioni e/o i software da testare.

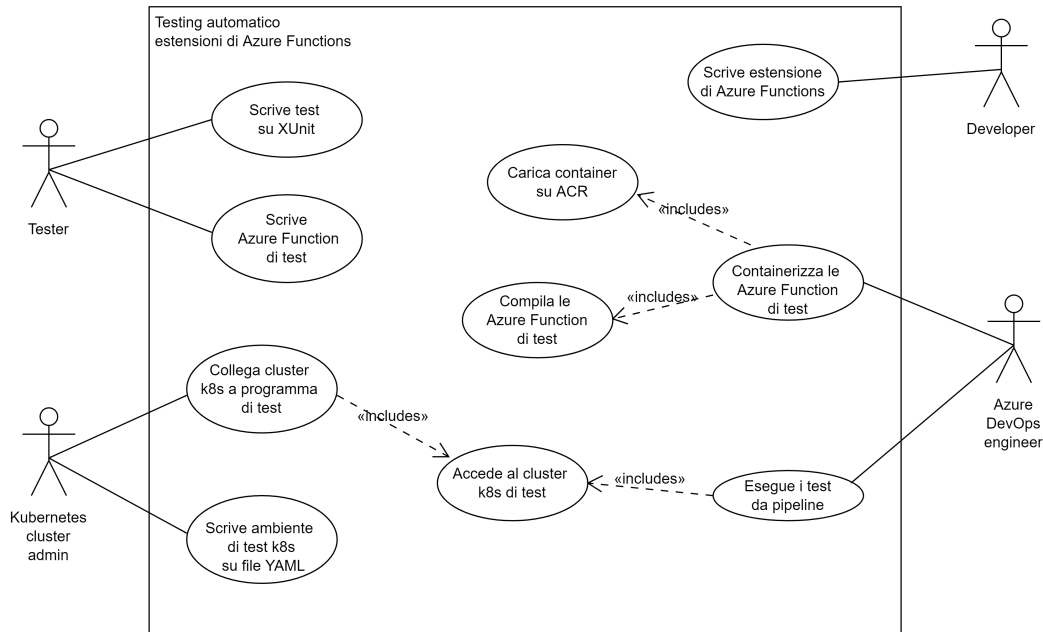


Figura 3.1: Diagramma UML *use cases* per un sistema di test automatico per Azure Functions su k8s

- **Tester:** implementano i test ed, eventualmente, le Azure Function di test, utilizzando le estensioni da testare. I test sono implementati in maniera tale da astrarre l'infrastruttura che sta eseguendo le Function (Azure, Kubernetes, Docker, ...).
- **amministratori del cluster k8s:** progettano e definiscono gli ambienti di esecuzione sottoforma di file YAML. Inoltre gestiscono l'accesso al cluster k8s dal programma di test. Da questo punto si ricavano due importanti requisiti: il sistema deve poter leggere da file sia le credenziali per l'accesso al cluster, sia le definizioni degli oggetti k8s in formato YAML da inoltrare al cluster.
- **Azure DevOps engineer:** hanno il compito di progettare, implementare ed eseguire le pipeline su Azure DevOps. In particolare queste pipeline devono containerizzare le Azure Function e mandare in esecuzione i test su di esse (test che richiedono l'accesso a Kubernetes).

Capitolo 4

Azure Functions e Kubesuite: progettazione e implementazione

Definiti i requisiti, il passo successivo è la progettazione dell'architettura. In questa fase l'obiettivo è quello di individuare le singole parti che compongono il prodotto finale e le relazioni tra questi.

4.1 Progettazione architetturale

La soluzione proposta al team di sviluppo è composta da due parti sostanzialmente indipendenti:

- una libreria, chiamata **Kubesuite**, che si occupa di gestire l'infrastruttura Kubernetes per il test. In particolare le funzionalità implementate sono:
 - caricamento degli oggetti k8s che compongono l'ambiente di test dai file YAML,
 - gestire l'avvio e la terminazione di questi nel cluster k8s,
 - estrarre il log dal pod da testare per passarlo ai metodi di test.
- una pipeline per Azure DevOps, da utilizzare come modello, cambiando soltanto il valore di alcune variabili, insieme a un nuovo Dockerfile per la containerizzazione delle Azure Functions.

4.1.1 Architettura di Kubesuite

Nell'immagine 4.1 è presente un diagramma di contesto (o *Data Flow Diagram* di livello 0) per la libreria Kubesuite. Kubesuite interagisce con un'interfaccia a linea di comando che manda in esecuzione i test e ne mostra all'utente i risultati, riceve dalle classi di test i metodi da eseguire e si collega al cluster k8s (tramite il file KUBECONFIG¹) per creare o terminare l'ambiente di test (dopo aver caricato le definizioni degli oggetti dai file YAML), leggere lo stato degli oggetti k8s in esecuzione e ricevere il log da inoltrare ai metodi di test.

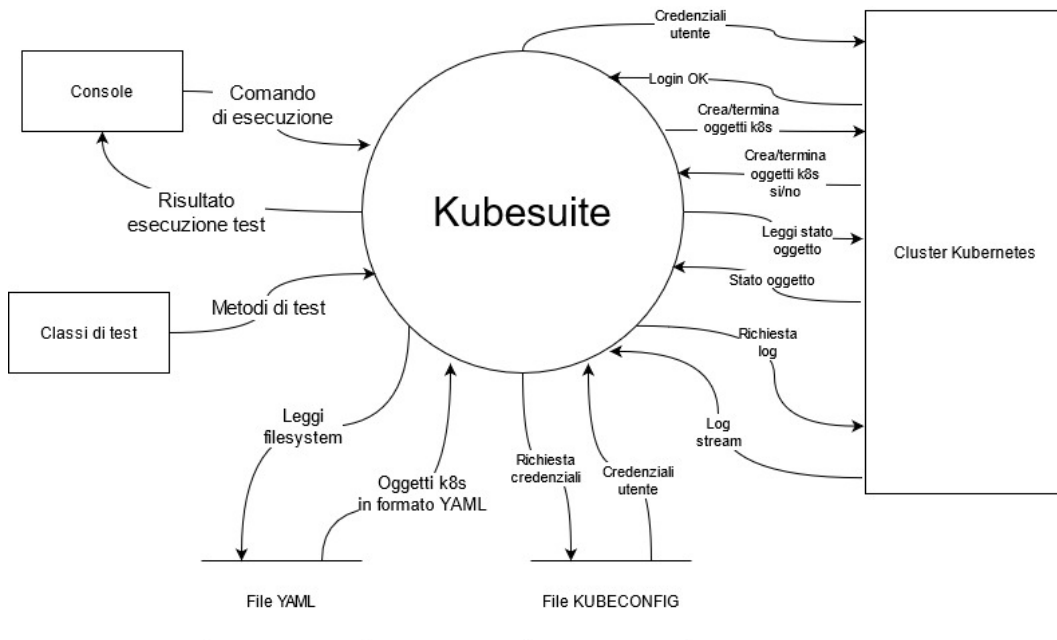


Figura 4.1: Diagramma di contesto di Kubesuite

Kubesuite è diviso in 6 moduli:

- un modulo centrale che ha compito di mandare in esecuzione i test,
- un modulo che raccoglie i risultati dei test per mostrarli sulla console,
- un modulo che si occupa di leggere i file YAML e convertire le stringhe YAML in oggetti,

¹Come già visto nella sezione 1.2.2, le credenziali per l'accesso ai cluster k8s sono memorizzate in dei file appositi. Di default il file si trova in `$HOME/.kubeconfig` oppure è nel path indicato nella variabile d'ambiente `KUBECONFIG`

- un modulo che si occupa di caricare le credenziali utente per inoltrarle al cluster,
- un modulo che si occupa della gestione degli oggetti k8s sul cluster, in particolare si occupa di avviarli e terminarli, e ne legge lo stato per capire quando sono operativi,
- un modulo che riceve il log dal Pod di test per inoltrarlo al metodo di test in esecuzione.

Tutto questo è visibile nel Data Flow Diagram di livello 1 in figura 4.2.

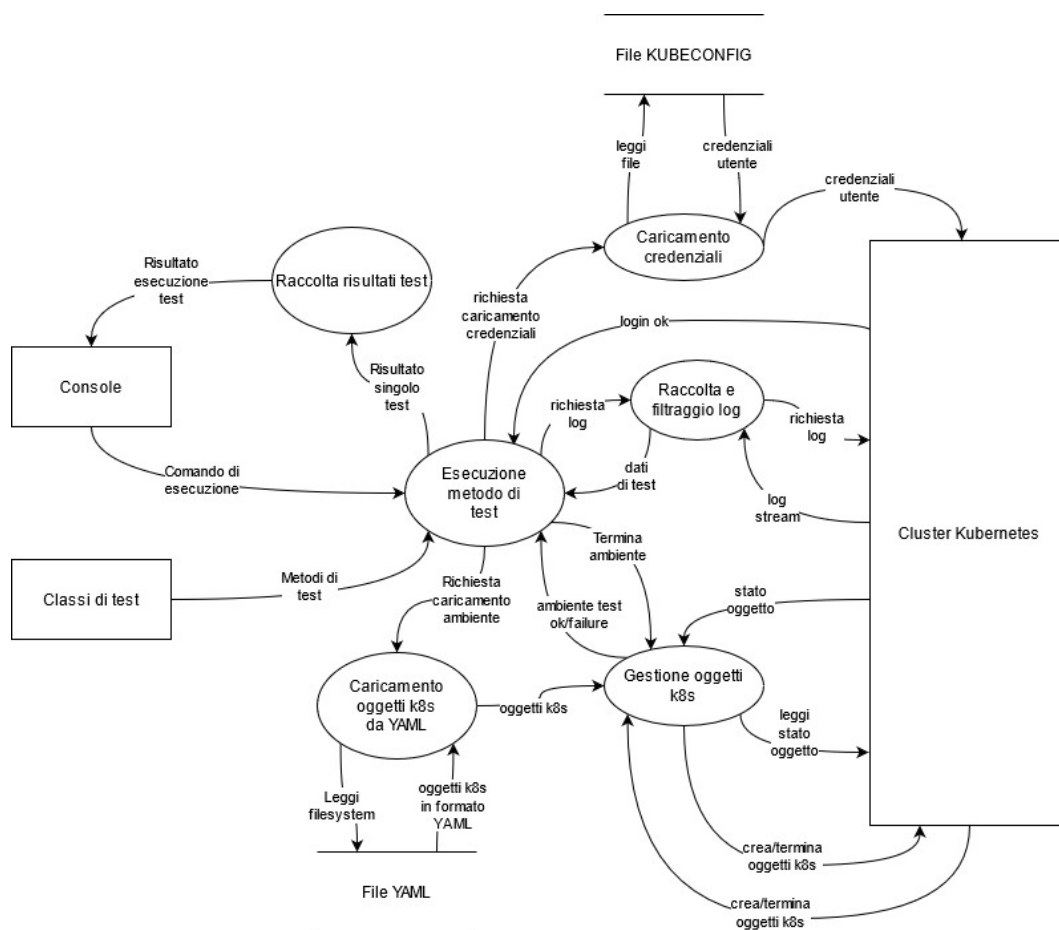


Figura 4.2: *Data Flow Diagram di livello 1 di Kubesuite*

4.1.2 Pipeline e Dockerfile

La pipeline è composta da tre job:

1. il primo scarica i sorgenti con il codice della Azure Function di test, scarica le dipendenze del progetto e lo compila. Fatto questo copia gli eseguibili e il Dockerfile nell'area di staging della pipeline. L'intero contenuto dell'area di staging viene poi pacchettizzato e pubblicato come *artifact*,
2. il secondo job scarica l'artifact ed esegue il comando **docker build** utilizzando il Dockerfile all'interno dell'artifact, copiando gli eseguibili della Function dall'artifact al container. Successivamente il container viene caricato su un container registry.
3. nel terzo e ultimo job viene clonato il progetto di test, installate le sue dipendenze e vengono poi eseguiti i test. Le credenziali per l'accesso al cluster k8s possono o essere già memorizzate nella macchina che esegue il job, oppure essere scaricate al momento.

4.1.3 Approfondimento: ricavare i dati di test dai Pod in esecuzione

I due principali problemi affrontati in questa fase sono come interfacciarsi a Kubernetes e come ottenere i dati da utilizzare nei test.

La soluzione adottata per il primo problema è stata quella di creare uno strumento ex-novo che si collega e gestisce sia il cluster di test (sfruttando le API Rest di Kubernetes) che l'esecuzione dei test da codice, in modo da aver maggior controllo sull'ambiente di test, dare la possibilità ai tester di utilizzare i framework di test (come xUnit) e avere maggiore integrazione con gli strumenti messi a disposizione da Azure DevOps (in particolare per quanto riguarda la raccolta dei risultati dei test).

Per quanto riguarda la seconda problematica gli approcci utilizzabili sono sostanzialmente due:

- confrontare lo stato iniziale dell'ambiente di test con quello ottenuto dopo l'esecuzione delle Function. Questo approccio è utilizzato, ad esempio, per verificare un collegamento a un database o a una coda di messaggi;

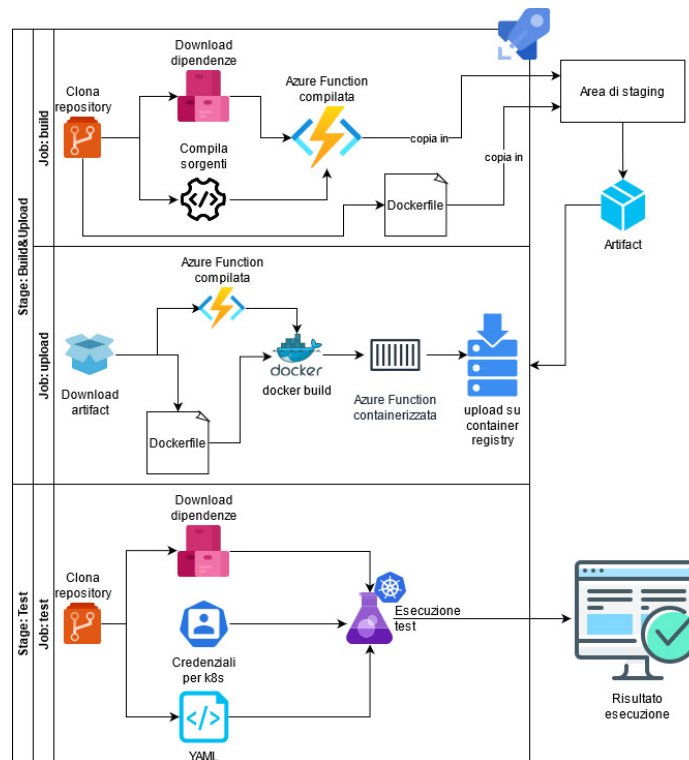


Figura 4.3: Diagramma pipeline CI/CD

- creare delle Function molto semplici che scrivono dei dati all'interno del log della App, filtrare il log per estrapolare i dati di esecuzione ed eseguire i test su questi. Questo metodo è utilizzato per verificare il funzionamento di trigger e input bindings e sarà quello utilizzato e analizzato nel proseguimento dell'elaborato.

4.2 Implementazione di Kubesuite

Kubesuite è una libreria scritta in C#/.NET e distribuita come pacchetto NuGet. Il suo scopo è quello di fornire ai tester una suite completa di metodi per scrivere dei test su Pod di Kubernetes in maniera trasparente rispetto all'infrastruttura. La libreria mette a disposizione le seguenti funzionalità:

- possibilità di definire gli oggetti necessari al test direttamente sui file YAML di Kubernetes: in questo modo il tester deve preoccuparsi solamente di istanziare gli oggetti sul cluster ed eseguire i test,

40 Azure Functions e Kubesuite: progettazione e implementazione

- interazione con Kubernetes direttamente dal codice: tutto il processo di test avviene all'interno del codice, non è necessario richiamare tool o righe di comando esterne (es. `kubectl`),
- robustezza del processo di test: la libreria è in grado di riconoscere gli errori sia in fase di deployment dell'ambiente che durante il test stesso. Nel caso questi avvengano il test viene abortito e il cluster riportato allo stato pre-test,
- flessibilità d'uso: la libreria mette a disposizione una serie di metodi standard, ma nulla vieta agli sviluppatori di aggiungere le loro implementazioni (soprattutto per quanto riguarda il collegamento al cluster), qualora fossero necessarie.
- trasparenza rispetto all'infrastruttura e all'ambiente utilizzato: incapsulando la gestione del cluster k8s all'interno di una classe, i metodi di test possono funzionare a prescindere dall'ambiente e dal cluster utilizzato.

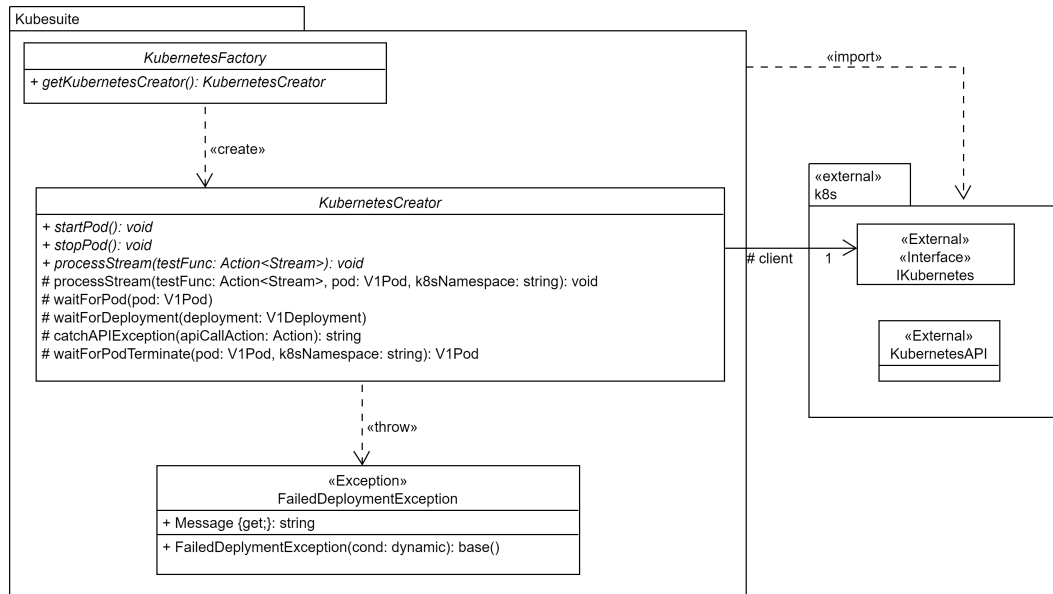
Kubesuite utilizza la libreria `KubernetesClient` versione 5.0.10 per collegarsi e manipolare i cluster k8s. `KubernetesClient` è la libreria client ufficiale per C#/.NET di Kubernetes.

`KubernetesClient` in realtà già fornisce quasi tutte le funzionalità presentate precedentemente, il fatto di costruirci sopra una nuova libreria nasce dall'esigenza di incapsulare delle procedure standard e risolvere alcuni problemi che si sono presentati durante l'implementazione dei test.

Per comodità, Kubesuite verrà diviso in due parti: la prima parte si occupa di gestire il cluster k8s, la seconda di caricare le definizioni degli oggetti Kubernetes dai file YAML.

La prima parte (rappresentata nel diagramma di classe in figura 4.4) è composta dalle classi:

- **`FailedDeploymentException`**: eccezione che viene lanciata quando un Pod o un Deployment falliscono in fase di avvio degli stessi,
- **`KubernetesFactory`**: classe astratta per l'implementazione del Factory pattern,
- **`KubernetesCreator`**: classe astratta, nonché principale interfaccia della libreria verso i programmatori. Contiene al suo interno i metodi per avviare e terminare correttamente gli oggetti Kubernetes utilizzati nei test, oltre a occuparsi di catturare il log dei Pod da passare come input

Figura 4.4: Diagramma UML della classe **KubernetesCreator**

ai metodi di test. Nello specifico i metodi astratti da implementare sono tre:

- **startPod()**, nel quale viene inserito il codice per avviare l’ambiente di test,
- **stopPod()**, nel quale gli oggetti avviati in **startPod()** vengono terminati,
- **processStream(Action<Stream>)**, nel quale viene estratto il log dai Pod creati in **startPod()** e passato alla procedura passata come argomento (tramite l’oggetto **Action<Stream>**, che rappresenta una procedura di tipo *void* che riceve come parametro un oggetto di tipo *Stream*). *Stream* è una classe della libreria di .NET che rappresenta un flusso di byte provenienti da una sorgente come un file, dispositivo di I/O o una socket. In questo caso la sorgente è il log del Pod che si vuole testare.

La seconda parte della libreria è composta dalle classi:

- **YamlTypeMap**: questa classe rappresenta delle coppie chiave-valore necessarie al parser di YAML.

- `YamlTypeMapBuilder`: builder per la classe `YamlTypeMap`,
- `YamlLoader`: classe di utility che converte le definizioni degli oggetti dai file YAML in oggetti, utilizzando una opportuna `YamlTypeMap` creata in precedenza.

4.2.1 Caricare le definizioni degli oggetti dai file YAML

Anche questa è una funzionalità già presente nella libreria `KubernetesClient`, tuttavia presenta alcune caratteristiche che necessitano di essere gestite correttamente:

1. il parser che converte le stringhe YAML in oggetti necessita che il programmatore indichi esplicitamente il tipo degli oggetti dichiarati nel file e il loro tipo corrispondente nel linguaggio di programmazione scelto. L'ordine di queste coppie tipo k8s - tipo oggetto deve corrispondere a quello delle dichiarazioni nei file YAML. In C# queste coppie devono essere memorizzate in un oggetto `Dictionary<string, Type>`, dove `string` è il tipo k8s come dichiarato nel file YAML, `Type` è la dichiarazione di tipo corrispondente nel namespace `k8s.Models` di `KubernetesClient`.
2. la lettura dei file avviene solo in modalità asincrona,
3. il parser torna gli oggetti caricati dal file come lista di tipi `object`, per cui è necessario effettuare il downcasting di tutti gli elementi al loro tipo effettivo.

Builder e `YamlTypeMap`

Per risolvere il problema al punto 1, la soluzione scelta è stata quella di utilizzare un altro pattern tipico dell'OOP: il *builder*.

Sfruttando il fatto che le possibili coppie stringa-tipo sono stabilibili a priori (ogni tipo Kubernetes ha il proprio tipo C# corrispondente), la lista di queste è stata incapsulata in un ADT chiamato `YamlTypeMap`.

Un oggetto `YamlTypeMap` è immutabile e può essere istanziato solo dal suo builder `YamlTypeMapBuilder`.

```
var typeMap = new YamlTypeMapBuilder()  
    .addPod()           //Aggiunge un pod
```

```
.addConfigMap()    //Aggiunge una configMap
.addSecret()       //Aggiunge un secret
.addDeployment()   //Aggiunge un Deployement
.addService()      //Aggiunge un Service
.build();          //Crea la YamlTypeMap
```

In questo frammento di codice sono presenti tutti i metodi pubblici di `YamlTypeMapBuilder`, con la loro funzione scritta nel commento accanto. Il builder è implementato in modo da usare il *fluent design*, ossia il fatto di poter concatenare tra loro le chiamate ai metodi di `YamlTypeMapBuilder` in un'unica espressione. Una volta chiamato il metodo `build()` non è più possibile modificare la `YamlTypeMap` ottenuta.

Gestione della lettura asincrona

Il fatto che il parsing dei file YAML venga svolto in modo asincrono, a prima vista sembra un vantaggio perchè aumenta il parallelismo del programma e, di conseguenza, le prestazioni dello stesso.

In C# l'esecuzione di un metodo asincrono comporta la sua interruzione (in attesa che l'operazione asincrona termini), mentre il resto del programma prosegue l'esecuzione.

Nel caso analizzato, invece, questa fattispecie rappresenta un problema: l'interruzione del metodo per il caricamento del file (da eseguire prima di ogni altra operazione) provocava l'esecuzione dei metodi di test da parte del runtime di .NET, con il risultato che questi fallivano perchè cercavano di ottenere delle informazioni da oggetti (ancora) inesistenti!

Per risolvere questo problema la soluzione adottata è stata quella di incapsulare l'operazione asincrona in un oggetto di tipo `Task<T>`² e utilizzare successivamente il campo `Result`.

In figura 4.5 è presente il diagramma UML che rappresenta questa seconda parte della libreria.

²`Task<T>` è la classe di .NET che rappresenta l'esecuzione di un metodo asincrono di tipo T. Richiamando il campo `Result` (il valore ritornato dal metodo) della Task in un metodo, il thread di questo viene interrotto finchè il thread della Task non termina, simulando così un'esecuzione in sequenza.

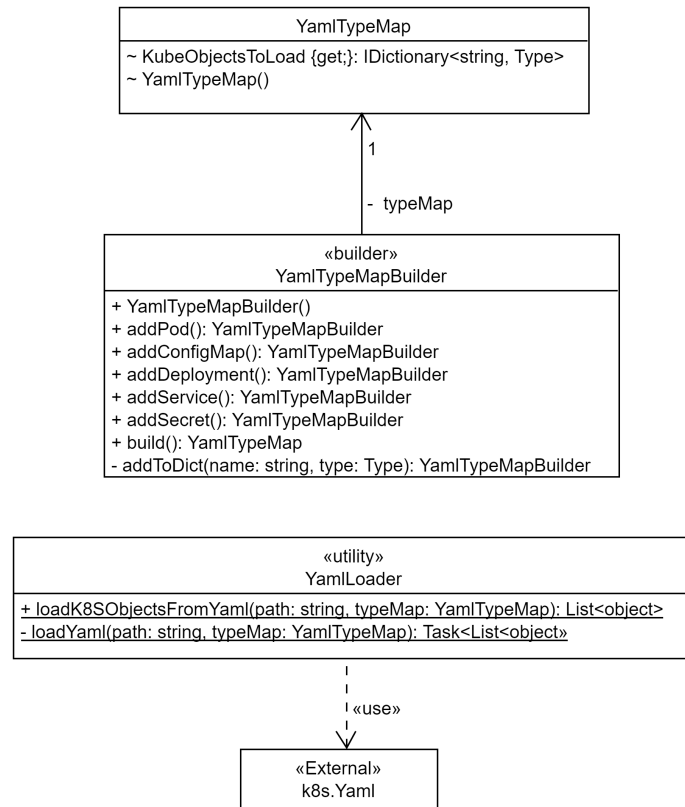


Figura 4.5: Diagramma UML della classe `YamlLoader`

4.2.2 Avviare l'ambiente di test

Una volta caricate le definizioni degli oggetti dai file YAML, il passo successivo è istanziarli nel cluster k8s utilizzando le opportune chiamate REST (già implementate in `KubernetesClient`) all'API Server di Kubernetes, senza che queste poi attendano che l'operazione richiesta sul cluster si concluda. Per semplicità si assuma che venga tutto creato all'interno dello stesso namespace k8s.

Se per `Service`, `Secret` e `ConfigMap` questa fattispecie non rappresenta un problema, perchè la loro creazione è praticamente immediata, lo stesso non avviene per `Pod` e `Deployment`, per cui si rende necessario attendere che questi siano operativi prima di proseguire con i test.

Per fare ciò il metodo migliore è quello di leggere periodicamente lo stato dell'oggetto dall'API Server finchè questo non raggiunge lo stato desiderato e, in caso di errore, individuarlo per poter fermare il tutto. In caso di er-

rore `KubernetesCreator` lancia l'eccezione `FailedDeployment`. Per ulteriori approfondimenti si rimanda alle appendici A.1 e A.2.

4.2.3 Estrarre il log ed eseguire il test

Una volta che l'ambiente è operativo, si può proseguire il test. L'approccio proposto è quello di implementare dei metodi usando il log di un Pod come parametro. Ricevere un log da un Pod è semplice: utilizzando il metodo `ReadNamespacedPodLog`, si riceve come risultato le stringhe del file di log di un Pod specifico come oggetto `Stream`. Se nella chiamata al metodo si aggiunge il parametro opzionale `follow: true` le righe del log vengono inviate in tempo reale al client.

Dal punto di vista di `KubernetesCreator`, questa operazione deve essere implementata nel metodo astratto `processStream`. Questo metodo ha due parametri:

- un riferimento a un metodo di tipo `void` con un solo parametro di tipo `Stream`. All'interno di questo metodo è implementato il test,
- un limite di tempo come oggetto `TimeSpan`. `TimeSpan` è una classe di C# che rappresenta intervalli di tempo.

Questo metodo torna un valore booleano: se il metodo passato al primo parametro non è terminato entro il tempo massimo indicato nel secondo parametro assume valore `true`, se invece il metodo è terminato correttamente assume valore `false`.

Dal punto di vista della classe di test, sono quindi necessari due metodi: nel primo si invoca il metodo `processStream` passandogli come parametro il riferimento al secondo metodo.

Il problema del log pendente

Considerato che nella grandissima maggioranza dei casi questa operazione si limita a un singolo Pod, è stato implementato un overload protetto di `processStream` che si occupa proprio di fare ciò per un Pod arbitrario passandogli come parametro. Così facendo l'implementazione di `processStream` nelle classi figlie di `KubernetesCreator` si riduce a una chiamata al metodo della classe padre, utilizzando uno dei Pod presenti tra i campi della classe come parametro attuale.

Lo **Stream** di un Pod ha una caratteristica peculiare: se ricevuto in modalità *follow* non è possibile stabilire a priori nè la sua lunghezza nè quando terminerà. Infatti sono possibili tre casi: il processo in esecuzione nel Pod termina, il Pod viene terminato forzatamente da Kubernetes, oppure il processo smette di scrivere sul log rimanendo in esecuzione. In quest'ultimo caso un'operazione di lettura sullo stream entra in attesa indefinita e può bloccare l'intero programma di test.

Per risolvere questo problema è stata implementata in **KubernetesCreator** questa procedura:

1. isolare l'operazione di estrazione ed analisi del log in un thread a parte: per fare ciò, la chiamata a **ReadNamespacedPodLog** è stata incapsulata in una **Task**,
2. impostare un timeout sul thread principale, utilizzando il limite di tempo passato passato a **processStream**.
3. a questo punto il test può terminare in due modi: o il timeout è scaduto o la **Task** è terminata (in questo caso il campo **IsCompleted** della **Task** è uguale a **true**). Solitamente, se il test termina per il primo caso, viene considerato fallito.

4.2.4 Terminare l'ambiente di test

Terminare l'ambiente di test significa eliminare dal cluster k8s tutti gli oggetti creati durante in fase di avvio. Questa operazione viene svolta in tre momenti: quando un test termina, quando viene lanciato un **FailedDeploymentException** e prima della chiamata a **startPod()** (non obbligatorio, ma consigliato nel caso in cui non si sia sicuri dello stato del cluster in pre-test). Per gli ultimi due casi sono necessari degli accorgimenti aggiuntivi:

- la fase di caricamento degli oggetti non può avvenire in **startPod()**,
- il metodo che implementa l'operazione, che in **KubernetesCreator** è **stopPod()**, deve essere abbastanza robusto da non bloccarsi nel caso tenti di eliminare un oggetto inesistente nel cluster perchè non ancora istanziato,
- nel terzo caso, così come era nell'avvio, l'eliminazione di Pod e Deployment non è immediata. Se per i Deployment non è un problema (perchè

ogni Pod di un Deployment ha un nome diverso), per i Pod il rischio è di tentare di creare Pod con lo stesso nome di un Pod che è in fase di eliminazione, con il risultato che la creazione fallisce per un problema di omonimia.

Per il primo problema la soluzione è quella di spostare la prima fase all'interno del costruttore di `KubernetesImplementation`.

Per il secondo problema la soluzione è inserire la chiamata ai metodi `DeleteNamespaced` di `KubernetesClient` in dei blocchi try-catch. Come già detto in sezione 4.2.2, i metodi di `KubernetesClient` non fanno altro che implementare delle chiamate REST. Quando una chiamata REST fallisce nella risposta è presente un codice HTTP di errore (40x o 50x). Quando ciò avviene i metodi di `KubernetesClient` lanciano l'eccezione `HttpOperationException`. Questa fattispecie è facilmente generalizzabile, perciò in `KubernetesClient` è stato implementato il metodo `CatchAPIExceptions` che riceve in input una funzione o un metodo *void* senza parametri, che preveda una o più chiamate alle API di Kubernetes. In caso di eccezione il metodo `CatchAPIExceptions` ritorna il messaggio contenuto nella risposta HTTP, altrimenti ritorna `null`. Per cui, dal punto di vista del chiamante, per riconoscere se si sono verificati errori è sufficiente verificare che il valore ritornato da `CatchAPIExceptions` sia `null`.

Per il terzo problema, concettualmente, la soluzione è molto simile a quella già vista per l'avvio dei Pod, ossia leggerne periodicamente lo stato finché non si verifica una certa condizione. In questo caso lo stato viene letto utilizzando `CatchAPIExceptions`. In questo modo nel momento in cui il Pod richiesto viene eliminato e se ne richiede lo stato, viene lanciata l'eccezione e la stringa ritornata dal metodo è diversa da `null`.

4.3 Implementazione della pipeline

Tutte le librerie su cui si basa Brainkin (estensioni e RuleEngine) sono inserite in un unico progetto Azure DevOps, chiamato `Beantech.SF`. Ad ogni libreria è associato un repository git su Azure Repos. I repository con le estensioni sono riconoscibili dal prefisso `Beantech.SF.Azure.Functions.Extensions`. La struttura di questi repository è riportata in figura 4.6.

Dal punto di vista dell'organizzazione del codice, un'estensione è una soluzione Visual Studio composta da 4 progetti C# (csproj):

- `Beantech.SF.Azure.Funcions.Extensions.<nome-estensione>` è l'estensione vera e propria: questa viene compilata come pacchetto NuGet e caricata sul feed privato `Beantech.SF.Nuget` hostato sull'Azure Artifacts del progetto Azure DevOps.
- `Beantech.SF.Azure.Funcions.Extensions.<nome-estensione>.Func` è la Function App di test: questa deve essere compilata, containerizzata e caricata su un container registry ACR (Azure Container Registry) privato.
- `Beantech.SF.Azure.Funcions.Extensions.<nome-estensione>.Func.Tests` è il progetto (basato su xUnit e Kubesuite) in cui vengono implementati i test dell'immagine docker della Function App di test. I test implementati in questo progetto devono essere eseguiti su pipeline di Azure DevOps in un cluster AKS (Azure Kubernetes Service) specifico per lo scopo.
- `Beantech.SF.Azure.Funcions.Extensions.<nome-estensione>.Tests` è il progetto (basato su xUnit) in cui sono implementati gli unit tests dell'estensione.

Il team di sviluppo di Brainkin ha già sviluppato la pipeline che compila, effettua lo unit testing sulle estensioni e carica il pacchetto NuGet su Azure Artifacts (scritta nel file `build-extension.yaml`), mentre devono essere scritti ex-novo il Dockerfile e la pipeline (scritta nel file `test-extension.yaml`) che compila, containerizza ed effettua i test sulla Function App. Entrambe le pipeline vengono eseguite su un agent on-premise (Beantech-Linux) presente presso la sede Beantech di Udine.

Tenendo conto di tutte queste caratteristiche qui descritte, sono necessari i seguenti strumenti di sviluppo, ognuno richiamabile da una task di Azure DevOps:

- la cli³ `dotnet` versione 3.1 per la compilazione dei sorgenti C# (richiamabile dalla task `DotNetCoreCLI`),
- la cli di Docker per containerizzare la Function App di test (richiamabile dalla task `Docker`),
- Azure CLI per scaricare le credenziali del cluster di test AKS (richiamabile dalla task `AzureCLI`).

³*cli*: abbreviazione di *command line interface* (interfaccia a linea di comando). In questo caso ci si riferisce alla cli come un programma sprovvisto di interfaccia grafica che viene eseguito utilizzando la shell di Linux.

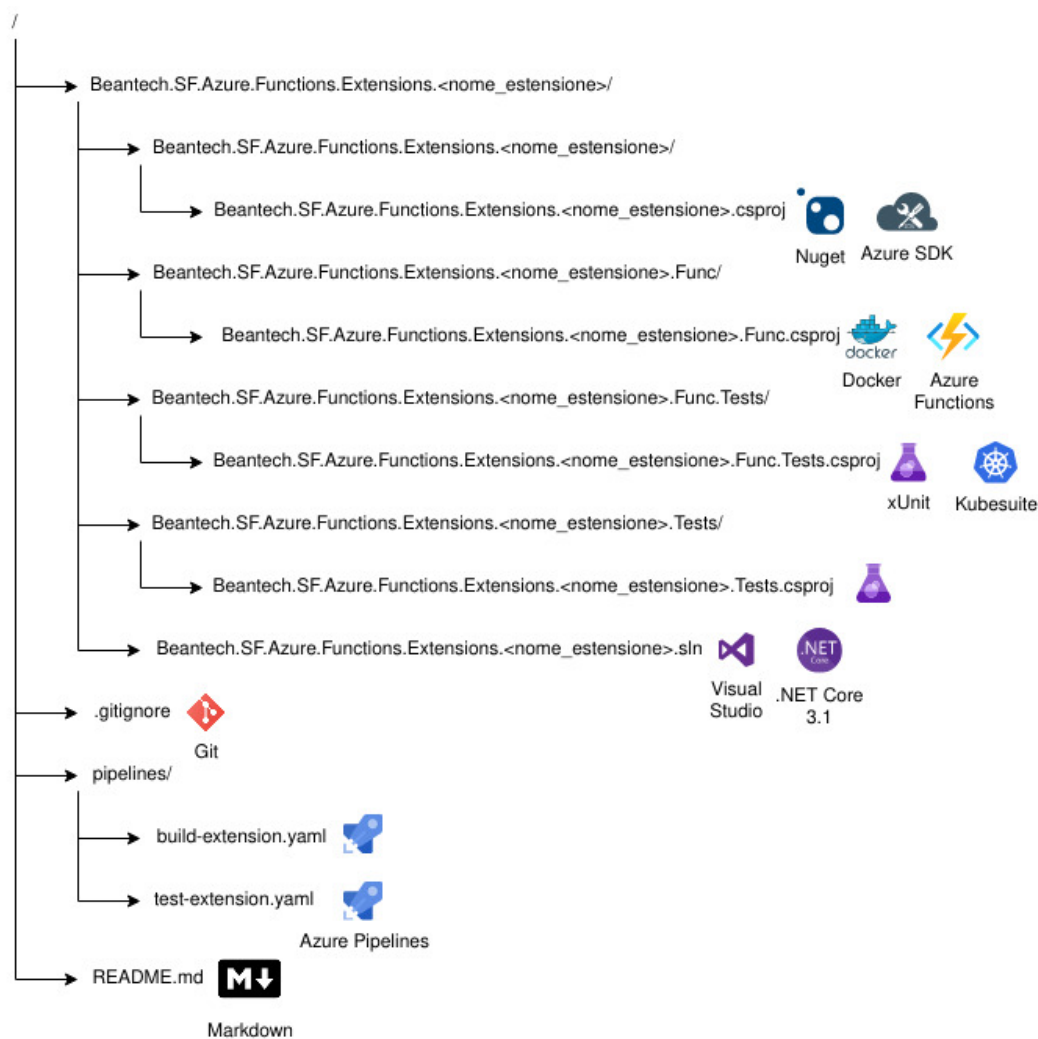


Figura 4.6: Struttura dei repository git delle estensioni per Brainkin. A lato sono visibili le tecnologie utilizzate per il progetto

Essendo queste pipeline eseguite su un agent di cui non è stato possibile stabilire, in assenza di un accesso alla shell, se tutti questi strumenti fossero già installati, la soluzione adottata è stata quella di eseguire *tutti* i job all'interno di container.

La pipeline ottenuta è consultabile in appendice A.6. Seguirà ora la descrizione dettagliata dei singoli job della pipeline raggruppati per stage. Nei titoli delle sezioni, dopo i due punti, è riportato il nome dello stage/job come scritto nel codice YAML e visibile nel report dei risultati.

4.3.1 Primo stage: Build Azure Function as Docker Image

Compilazione dei sorgenti: Build

Per compilare i sorgenti delle Function normalmente si utilizza il comando `dotnet publish <path-del-file-csproj> -o <path-degli-eseguibili>`, ottenendo come output una directory con la struttura come quella mostrata in figura 1.4. Il contenuto di questa directory deve essere poi copiato nel container.

Il comando `publish` esegue implicitamente `dotnet restore` e `dotnet build`. Il primo installa le dipendenze del progetto da NuGet, il secondo compila i sorgenti in eseguibili `.dll`, alla fine `publish` prende le dipendenze e gli eseguibili e li inserisce tutti in nella stessa directory, in modo che l'output di compilazione ottenuto sia facilmente trasportabile su macchine diverse senza dover reinstallare ogni volta le dipendenze.

Normalmente questo passaggio viene svolto direttamente nel Dockerfile utilizzando il template messo a disposizione da Visual Studio per le Azure Functions containerizzate, tuttavia questo template non è utilizzabile nelle pipeline di Azure DevOps nel caso i pacchetti NuGet da scaricare siano presenti su feed di Nuget privati, come in questo caso. Questo avviene perchè per effettuare l'accesso ai feed privati di Azure Artifacts bisogna ottenere un token OAuth facendo l'accesso da un browser (non fattibile da un Dockerfile), oppure indicando un PAT (Personal Access Token) nel file `nuget.config` della soluzione (in cui vengono elencati i feed utilizzati dal progetto).

Questa seconda soluzione è quella utilizzata inizialmente dal team di sviluppo, ma presenta due problemi: i PAT hanno durata limitata (per cui a PAT scaduto si rendeva necessario generarne uno nuovo e correggere tutti i

repository) e sono associati all'utente, presentando un problema di sicurezza non indifferente: nel caso in cui un estraneo riuscisse a entrare in possesso di un PAT troppo permissivo, avrebbe accesso alle risorse accessibili dall'account (repository, feed, pipelines, ecc...)!

Per risolvere questo problema, la fase di compilazione dei sorgenti è stata spostata dal Dockerfile in un job apposito nella pipeline, composto da 4 step:

1. **restore**: questo step scarica le dipendenze del progetto. Utilizzando l'opzione `feedsToUse: select` la task genera temporaneamente un nuovo `nuget.config` con il feed Azure Artifacts indicato in `vstsFeed`;
2. **publish**: esegue il comando `dotnet publish`. In questo caso la directory di output è indicata nella variabile built-in `$(Build.ArtifactStagingDirectory)`.
3. un semplice script bash per copiare il Dockerfile dal repository a `$(Build.ArtifactStagingDirectory)`
4. il contenuto della directory `$(Build.ArtifactStagingDirectory)` viene pubblicato come artifact della pipeline, in modo da poter essere utilizzato nei job successivi. Un esempio del contenuto di questo artifact è rappresentato nel diagramma in figura 4.7.

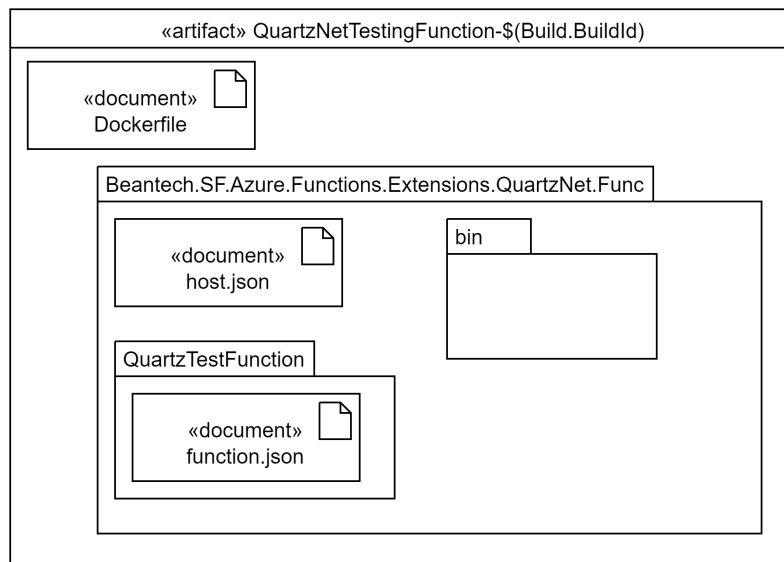


Figura 4.7: Contenuto degli *artifact* pubblicati dal job Build

52 Azure Functions e Kubesuite: progettazione e implementazione

Così facendo non c'è il rischio che le credenziali del feed NuGet privato vengano esposte verso l'esterno, in quanto memorizzate direttamente da Azure DevOps.

Questo job è stato eseguito all'interno di un container con immagine Docker `mcr.microsoft.com/dotnet/core/sdk:3.1`, ossia l'immagine ufficiale di Microsoft nella quale è già preinstallato .NET Core SDK 3.1 e la cli `dotnet`.

Dockerfile e build dell'immagine Docker: Docker

Per containerizzare una Function App si utilizzano come immagini di partenza i runtime di Azure Functions in base al linguaggio di programmazione utilizzato, tutte disponibili su Docker Hub. Per C#/.NET l'immagine di partenza utilizzata è `mcr.microsoft.com/azure-functions/dotnet:3.0`. Successivamente vanno impostate due variabili d'ambiente per il container:

- `AzureWebJobsScriptRoot`: indica il path in cui sono presenti i file di configurazione e gli eseguibili della Function App, in questo caso la directory `/app` creata appositamente,
- `AzureFunctionsJobHost__Logging__Console__IsEnabled`: se impostata a `true` abilita il logging.

L'ultimo passo è quello di copiare gli eseguibili compilati dal file system dell'agent alla directory `/app` nel container.

Per quest'ultimo passaggio la directory del sistema host da cui copiare i file non è stata riportata hard-coded all'interno del Dockerfile, ma definita da una variabile `PUBLISH_DIR`, che viene passata al comando `docker build` con il flag `--build-arg PUBLISH_DIR=<path-directory>`.

All'interno della pipeline questi passaggi sono stati racchiusi in un deployment job, con strategia `runOnce`, eseguito in un container di Ubuntu 20.04:

1. come da specifiche dei deployment job, viene scaricato l'artifact pubblicato precedentemente nella directory `$(Pipeline.Workspace)`, che è la directory di lavoro principale della pipeline all'interno dell'agent,
2. si procede con l'installazione della cli di Docker (non presente nell'immagine di partenza),

- viene fatto il login al container registry su Azure. Anche in questo caso le credenziali sono state memorizzate in precedenza da Azure DevOps, per cui è sufficiente riportare il nome del registry alla task **Docker**,
- si esegue il build dell'immagine usando come Dockerfile quello presente all'interno dell'artifact scaricato al punto 1 e indicando come **PUBLISH_DIR** la directory dell'artifact con all'interno la Function App compilata.
All'immagine ottenuta viene assegnato il seguente nome:
`beantech.sf.services.function.<nome-estensione>`,
- infine si esegue il push dell'immagine sul container registry, assegnando all'immagine il tag **latest**.

4.3.2 Secondo stage: Test Azure Function

Esecuzione dei test su pipeline: Test

Per eseguire quest'ultima fase del processo è stato necessario creare un'immagine Docker apposita, basata su Ubuntu 20.04, al cui interno sono installati Azure CLI, .NET Core 3.1 SDK, **kubectl** e **helm**. Questi ultimi due programmi servono a interagire con Kubernetes da script bash in caso di necessità. In questo caso non saranno utilizzati. Il Dockerfile dell'immagine è consultabile in appendice A.4.

Il job utilizzato è composto da quattro step:

- download delle credenziali al cluster AKS da Azure. Per fare ciò si utilizza la task **AzureCLI**, che permette di eseguire comandi sulla shell di Azure. Anche in questo caso le credenziali per l'accesso ad Azure sono memorizzate da Azure DevOps. Per ottenere le credenziali di Kubernetes viene eseguito il comando

```
az aks get-credentials --resource-group  
→ <nome-del-resource-group-di-Azure> --name  
→ <nome-del-cluster-aks>
```

Un *resource group* di Azure è un contenitore di risorse di Azure come macchine virtuali, container registry ACR, cluster AKS, Azure Function Apps. A ogni risorsa è associata un resource group e ogni resource group è associato a una sottoscrizione, a cui viene poi addebitato il costo delle risorse usate mensilmente.

54 Azure Functions e Kubesuite: progettazione e implementazione

2. impostazione della variabile d'ambiente `KUBE_PATH`: questa variabile servirà poi al programma di test per indicare la directory in cui sono contenuti i file YAML di Kubernetes con l'ambiente di test,
3. restore della dipendenze per il progetto di test (Kubesuite e xUnit) dal feed NuGet `Beantech.SF.NuGet`
4. esecuzione dei test con la task `DotNetCoreCLI` e il comando `dotnet test`. A test conclusi la task raccoglie i risultati e li mostra su un'interfaccia grafica di riepilogo.

Capitolo 5

Testing automatico di Azure Function con Kubesuite

In questo capitolo saranno analizzate alcune tecniche di testing automatico di estensioni per Azure Function utilizzando Kubesuite. Dopo l'analisi sulla struttura e il funzionamento sia delle Function di test che dei metodi di test, verranno illustrati di due esempi di test funzionanti.

5.1 Struttura e funzionamento di una Function di test

Per testare il corretto funzionamento di una estensione per Azure Function, il metodo più semplice è creare una o più Azure Function, che ricevono i dati dal trigger (o dall'input binding) da testare, li elabora e li restituisce sul log della Function App per essere utilizzati dal metodo di test.

In C#/.NET le Function sono inserite all'interno di classi statiche¹, mentre il codice è scritto all'interno di un metodo contraddistinto, dall'attributo `[FunctionName("nomeDellaFunction")]`, che serve a indicare il nome con cui la Function è identificata all'interno della Function App.

Gli attributi di C# sono dei metadati aggiuntivi che vengono associati a classi, metodi, parametri o variabili, che vengono aggiunti in fase di compilazione e possono essere interrogati in fase di runtime. Non sono da confondere con gli attributi (o campi) delle classi.

¹Una classe statica è una classe che non può essere istanziata

I metodi delle Function accettano un numero variabile di parametri, tra questi almeno uno deve rappresentare un trigger, mentre l'ultimo deve essere di tipo `ILogger`, tutti gli altri sono i bindings.

L'interfaccia `ILogger` è un'astrazione per un dispositivo di output testuale verso cui inviare le stringhe di log. Nel caso delle Function App containerizzate, l'interfaccia astrae lo standard output del container, il quale viene poi inviato al logging driver del runtime di Docker/Kubernetes.

Trigger e bindings della Funzione possiedono tutti un attributo, nel quale vengono definiti i suoi parametri. Questi possono essere di due tipi: costanti, oppure, utilizzando la sintassi `"%variabile%"` possono accedere alle variabili d'ambiente del container/pod in cui sono in esecuzione. Quest'ultima opzione deve però essere prevista dall'implementazione dell'attributo.

Per stampare le stringhe con i dati di test si utilizza il metodo `LogInformation` di `ILogger`.

Per raccogliere i dati di test il metodo più efficace è quello di utilizzare degli oggetti molto semplici, composti solo da una serie di campi (in base ai dati di interesse), da serializzare in stringhe JSON, stampate in una riga del log. Queste poi saranno deserializzate nell'oggetto di partenza all'interno del metodo di test. Per serializzare e deserializzare oggetti in e da JSON sono disponibili molte librerie per C#, sia native che di terze parti.

Costruttore e distruttore delle classi di test

Quando si esegue il comando `dotnet test` in un progetto xUnit.NET, il framework di test individua i metodi con attributo `[Fact]`. Prima di eseguire un metodo di test, xUnit istanzia la classe del metodo e, quando il metodo di test termina, l'istanza viene deallocata.

Sfruttando questo fatto, la fase di avvio dell'ambiente di test viene svolta all'interno del costruttore della classe di test, mentre la fase di terminazione viene svolta all'interno del metodo `Dispose`.

Quest'ultimo è un metodo dell'interfaccia `IDisposable` e viene invocato insieme al Garbage Collector di .NET in fase di deallocazione di un oggetto. Il suo scopo è, principalmente, quello di rilasciare tutte le risorse non gestite direttamente dall'oggetto, in questo caso gli oggetti k8s creati invocando `startPod` nel costruttore, mentre nell'implementazione di `Dispose` ci si limita a invocare `stopPod`. Si rimanda all'appendice A.3 per ulteriori dettagli.

Filtrare il log delle Azure Functions

Il runtime delle Function App, dal punto di vista del logging, è molto verboso, motivo per cui è necessario implementare un meccanismo efficiente per filtrare il contenuto delle righe, in modo da conservare solo le stringhe di interesse.

Il log delle Azure Function è strutturato su due colonne: nella prima "colonna" è visualizzato il livello di logging del messaggio, mentre nella seconda colonna, invece, sono scritti i messaggi veri e propri.

I messaggi di log sono poi stampati su due righe: nella prima riga viene indicato il livello di logging e il thread che sta stampando il messaggio, nella seconda riga c'è il messaggio vero e proprio. Per la seconda riga, nella prima colonna sono presenti degli spazi vuoti per allineare verticalmente le due righe. Tra i messaggi di log sono presenti le stringhe JSON da filtrare con i dati di test. Questa struttura è visibile in figura 5.1.

Il log viene estratto, letto e filtrato una riga per volta. Si rimanda all'appendice A.2 per ulteriori dettagli.



Figura 5.1: Struttura del log di una Azure Function App

Una delle caratteristiche principali di JSON è la sua grammatica estremamente semplice, al punto che è possibile riconoscere una stringa JSON direttamente con un'espressione regolare. Infatti la sua grammatica context-free, così come riportata nel sito ufficiale [15] è facilmente trasformabile in una grammatica lineare, inoltre gli stessi autori del sito hanno messo a disposizione in forma grafica le macchine a stati finiti che riconoscono i lessemi delle stringhe JSON.

Esistono parecchie espressioni regolari per riconoscere le stringhe JSON, più o meno complesse a seconda delle esigenze. In questo caso, vista la struttura del log, per filtrare le stringhe JSON è sufficiente riconoscere le stringhe che

cominciano con una parentesi graffa aperta e terminano con una parentesi graffa chiusa. L'espressione regolare PCRE che descrive questo pattern è:

$$\backslash\{(.|\backslash s)^*\backslash\}$$

Dove il metacarattere `.` indica "qualsiasi carattere eccetto gli spazi bianchi", mentre `\s` indica i caratteri ASCII mostrati a schermo come spazi bianchi (spazio, fine linea, TAB, ESC, ...), la `*` è la star di Kleene.

Un altro aspetto utilizzato nei test è la conta delle occorrenze: ogni volta che viene riconosciuta una stringa JSON dal log si incrementa un contatore. In questo modo è possibile limitare la quantità di informazioni sufficienti per effettuare il test, soprattutto se il test prevede di contare quante volte una Function viene eseguita, oppure se questa viene attivata ripetutamente nel tempo.

5.2 Kubesuite: esempi d'uso

In questa sezione verranno presentati due esempi di test di funzionamento su due estensioni di Azure Functions per Brainkin, utilizzando Kubesuite e xUnit, nonché i risultati ottenuti dall'esecuzione di questi su pipeline.

5.2.1 Quartz.NET

Quartz.NET è una libreria open-source di scheduling temporale per .NET, che supporta le cron expression di Unix per definire quando un certo thread debba cominciare la sua esecuzione. In figura 5.2 è visualizzato il diagramma UML delle classi utilizzate per l'implementazione del test.

Attributi

Il team di Brainkin ha utilizzato questa libreria per costruire un trigger con i seguenti attributi:

- **CronScheduleExpression** (obbligatorio): cron expression del trigger. Una cron expression è una stringa composta da 6 o 7 campi separati da spazi. Nello specifico, da sinistra a destra: secondi, minuti, ore, giorno del mese, mese, giorno della settimana, anno. Tramite dei metacaratteri è possibile comporre degli scheduling complessi.

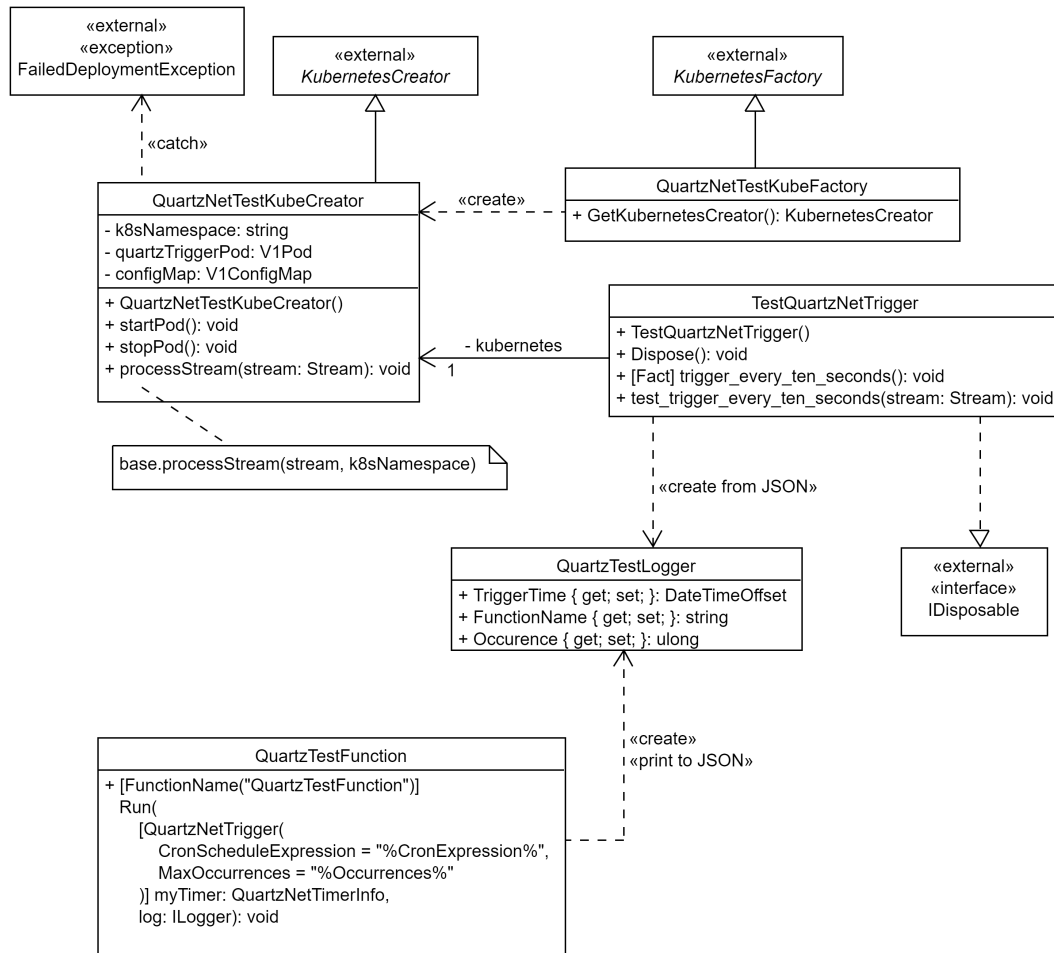


Figura 5.2: Diagramma di classi UML per il test su Quartz.NET

- **StartTimeUtc** e **EndTimeUtc**: datetime di inizio e/o fine dell'esecuzione del trigger in modo che la Function venga eseguita solo durante un certo intervallo di tempo definito.
- **MaxOccurrences**: un numero massimo di volte in cui viene eseguito il trigger.
- **Priority**: la priorità della Function: se due o più Function devono essere eseguite allo stesso momento, vengono eseguite a partire da quella a priorità più alta a quella a priorità più bassa.

Il trigger è passato alla Function come oggetto di tipo `QuartzNetTimerInfo`, con due campi:

- **CronExpression**: la cron expression del trigger,
- **CurrentOccurrenceNo**: il numero di volte in cui è stato attivato il trigger.

Ambiente di test

L'ambiente di test su Kubernetes è composto da:

- un Pod, chiamato `quartznet-trigger-test-azfunction`, composto da un container basato sull'immagine Docker `beantech.sf.services.function.quartztrigger:latest`, disponibile nel container registry `bkacr002dev.azurecr.io`.

All'interno di questo container sono istanziate le variabili d'ambiente `Occurrences` e `CronExpression`, i cui valori vengono passati dalla `ConfigMap quartznettrigger-test-configmap`.

- una `ConfigMap`, chiamata `quartznettrigger-test-configmap`, composta da due valori:
 - **Occurrences**: `"10"`, indica che la Function di test viene eseguita al più 10 volte,
 - **CronExpression**: `"*/10 * * * * ?"`, cron expression che indica che la Function di test deve essere eseguita ogni 10 secondi.

Il file YAML completo è consultabile in appendice B.1

Il test

Il test verifica, su 10 esecuzioni della Function di test, che ognuna venga eseguita a distanza di 10 secondi dalla precedente. Per fare ciò si prende data e ora della prima occorrenza come punto di partenza. Dalla seconda in poi si fa la differenza tra il tempo dell'occorrenza corrente con il tempo della precedente e si verifica che questa sia all'interno di un range di tempo accettabile.

Dai test effettuati in locale (e non su Kubernetes) questo intervallo di tempo oscilla tra gli 8.5 s e i 10.5 s. Queste fluttuazioni di tempo possono essere dovute a molteplici fattori: non determinismo dello scheduler del sistema host, ritardo tra l'esecuzione del trigger e l'invocazione della Function, overhead di esecuzione dovuto al container, risoluzione del metodo `DateTimeOffset.UtcNow` (circa

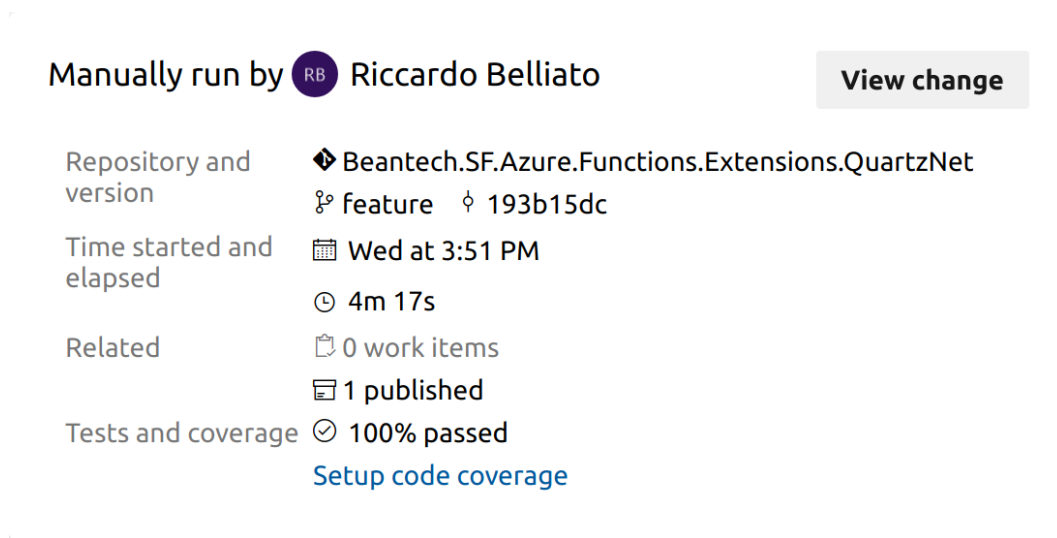


Figura 5.3: Riepilogo esecuzione pipeline per Quartz.NET

15-16 millisecondi) ed è molto probabile che dipendano anche dalla macchina in cui è eseguita la Function.

Le fluttuazioni sono più marcate tra la prima e la seconda esecuzione. Dalla seconda esecuzione in poi, il tempo tra una esecuzione e l'altra tende a stabilizzarsi tra 9.90s e 10.1s.

Per calcolare la differenza si utilizza il campo `TriggerTime` di `QuartzTestLogger`. Quando si fa la differenza tra due oggetti `DateTime` il risultato è ottenuto come oggetto `TimeSpan`. Tramite il metodo `TotalSeconds` la differenza viene restituita in secondi decimali come numero `double`.

Risultati

Si riportano ora i risultati e i log dell'esecuzione della pipeline `Beantech.SF.Azure.Functions.Extensions.QuartzNet.TestingFunction` di Azure DevOps con Build Id 10182, eseguita in data 28 luglio 2021. In figura 5.3 è stato riportato il riepilogo sull'esecuzione della pipeline direttamente dalla UI di Azure DevOps.

In figura 5.4 si osserva che entrambi gli stage sono stati eseguiti correttamente e come previsto: il primo ha pubblicato un artifact, il secondo ha eseguito il test dando esito positivo. Stesso discorso per i singoli job, tutti e

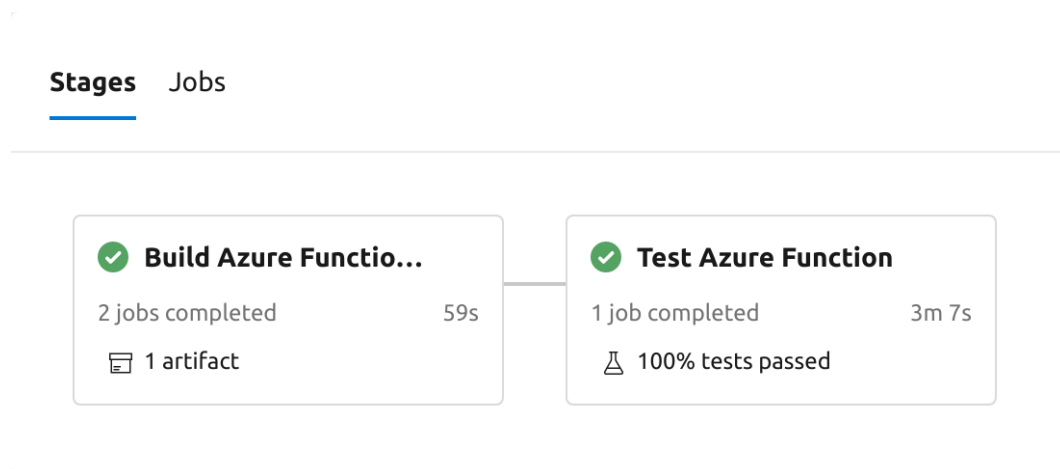


Figura 5.4: Riepilogo esecuzione stage per la pipeline di Quartz.NET

tre conclusi con esito positivo, come illustrato in figura 5.5.

The screenshot shows the 'Jobs' tab in Kubesuite. It displays a table with the following jobs:

Name	Duration
✓ Build Azure Function as Docker Imager: Build	⌚ 25s
✓ Build Azure Function as Docker Imager: Docker	⌚ 19s
✓ Test Azure Function: Test	⌚ 3m 3s

Figura 5.5: Riepilogo esecuzione job per la pipeline di Quartz.NET

Per quanto riguarda il terzo job, ne era presente un solo test, che si è concluso correttamente, come riportato in figura 5.6.

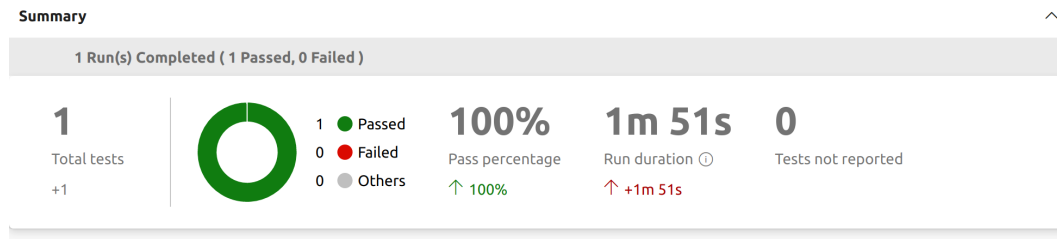


Figura 5.6: Riepilogo test per Quartz.NET

TriggerTime	secondi dal TriggerTime precedente
2021-07-28T13:53:50.0929978+00:00	
2021-07-28T13:54:00.0019332+00:00	9.9089354
2021-07-28T13:54:10.0007633+00:00	9.9988301
2021-07-28T13:54:20.0015726+00:00	10.0008093
2021-07-28T13:54:30.0010672+00:00	9.9994946
2021-07-28T13:54:40.0009354+00:00	9.9998682
2021-07-28T13:54:50.0007985+00:00	9.9998631
2021-07-28T13:55:00.0009757+00:00	10.0001772
2021-07-28T13:55:10.0006832+00:00	9.9997075
2021-07-28T13:55:10.0006832+00:00	10.0002064

Tabella 5.1: TriggerTime del test di Quartz.NET e differenza tra tempi consecutivi

Verranno ora analizzati i singoli tempi riportati nel log in appendice B.2 e calcolata la differenza a due a due per verificare il corretto funzionamento del test. Nella prima colonna è riportata la data e l'ora in formato ISO 8601. Come si può notare in tabella 5.1, le differenze tra i tempi sono in linea con le considerazioni fatte in precedenza, con valori molto vicini ai 10 secondi. Si noti anche la differenza di tempo tra il primo e il secondo **TriggerTime**, che si discosta maggiormente dalla media rispetto agli altri, media che si attesta sui 9.989 765 76 s, quindi pienamente in linea con i tempi previsti dal test.

5.2.2 MQTT

MQTT (Message Queue Telemetry Transport) è un protocollo basato sulla suite di protocolli TCP/IP, molto utilizzato in ambito IoT per la sua sempli-

cità e leggerezza. Si basa sul paradigma publish-subscribe, per cui per il suo funzionamento è necessario:

- un *broker*, ossia un server che ha il compito di ricevere e inviare i messaggi da e verso i client collegati,
- uno o più *publisher*, ossia dei client che inviano messaggi al broker insieme a un certo *topic* (messaggio di tipo *publish*),
- uno o più *subscriber*, ossia dei client che ricevono dal broker i messaggi associati solo a certi topic di interesse. Per fare ciò il client invia al broker un messaggio (*subscribe*) nel quale indica a quali topic intende iscriversi.

In figura 5.7² è illustrato in breve il funzionamento del protocollo.

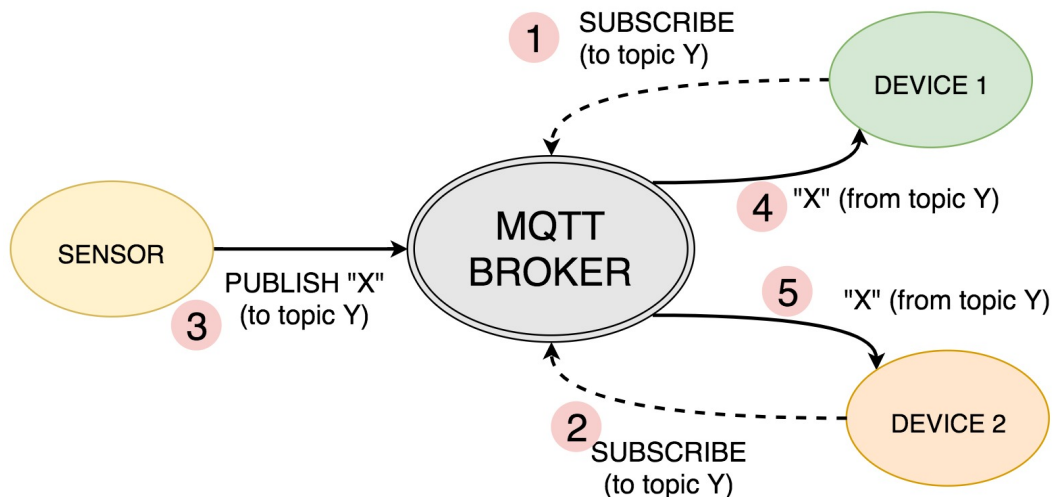


Figura 5.7: Funzionamento di MQTT

Il team di Brainkin ha costruito tre estensioni di Azure Functions che implementano il protocollo MQTT utilizzando la libreria MQTTnet³: un trigger (*MqttTrigger*), il quale rende la Function un *subscriber* passandole il messaggio MQTT come parametro, e due output binding:

- *Mqtt*: le opzioni dei messaggi MQTT (topic, QoS e retain) vengono impostati nell'attributo e non sono modificabili,

²Immagine tratta da <https://www.norwegiancreations.com/2017/07/mqtt-what-is-it-and-how-can-you-use-it/>

³<https://github.com/chkr1011/MQTTnet>

- `MqttMessage`: per ogni messaggio MQTT vanno indicate tutte le opzioni.

L'utilizzo di uno di questi output binding rende la Function un publisher.

Lo scopo di questo test è dimostrare il funzionamento di entrambi: una prima Function (`PublisherFunction`) avente `Mqtt` come output bindings deve pubblicare una serie di messaggi su un certo topic MQTT, mentre una seconda Function (`RunTriggerTopic01`) avente `MqttTrigger` come trigger e iscritta allo stesso topic deve ricevere i messaggi pubblicati dalla prima Function.

`PublisherFunction` è programmata come Task asincrono, infatti l'operazione di pubblicazione avviene chiamando un metodo asincrono e solo una volta che questo è terminato, viene stampato il JSON sul log.

In figura 5.8 è visualizzato il diagramma UML delle classi utilizzate per l'implementazione del test.

Attributi

- `ConnectionStringSettingsName`: indica il nome della variabile d'ambiente in cui sono riportate le informazioni per la connessione al broker sottoforma di stringa avente il seguente formato:
"Server=***;Port=***;Username=***;Password=***". All'occorrenza è possibile rimuovere gli ultimi due campi se il broker non richiede autenticazione. Se non indicata esplicitamente, l'espressione cerca la variabile d'ambiente `MqttConnection`.
- `Topic` (obbligatorio): il topic MQTT su cui pubblicare o iscriversi, questo campo non è impostabile tramite variabili d'ambiente, ma solo come costante.
- `QoSName`: il Quality of Service con cui pubblicare i messaggi verso il broker. Da specifiche MQTT esistono tre tipi di QoS:
 - At Most Once (0): il publisher invia il messaggio al broker una sola volta senza ulteriori passaggi per verificare la consegna,
 - At Least Once (1): il publisher o il broker riprovano a inviare il messaggio finchè questo non viene ricevuto correttamente. In questa modalità è possibile ricevere messaggi duplicati.
 - Exactly Once (2): il publisher o il broker si assicura che il ricevente riceva solo una copia del messaggio.

Con QoS 1 o 2 è garantito anche l'ordine dei messaggi: se un publisher pubblica due messaggi A e B in quest'ordine, i subscribers devono ricevere i due messaggi A e B in sequenza.

- **RetainName**: un messaggio MQTT flag **Retain** a 1, indica al broker di conservarlo finché non arriva un altro messaggio dello stesso topic e flag **Retain** a 1. In questo modo un eventuale client che si iscrive al topic in un secondo momento riceve subito l'ultimo messaggio pubblicato.

Raccolta dati di test

Il messaggio MQTT pubblicato da `PublisherFunction` è "Executing MQTT at <data e ora attuale>", con QoS 2 e flag **Retain** a 1. Il topic di pubblicazione è "test", il quale viene salvato in una costante `Topics.TOPIC_TEST`.

`PublisherFunction` ha come trigger un `QuartzNetTrigger` impostato ogni 5 s.

Ambiente di test

L'ambiente di test è leggermente più complesso rispetto a quello utilizzato per Quartz.NET e si compone di:

- un Pod, chiamato `mqtttrigger`, composto da un container basato sull'immagine Docker `beantech.sf.services.function.mqtttrigger:latest`, disponibile nel container registry `bkacr002dev.azurecr.io`.

All'interno di questo container sono istanziate le variabili d'ambiente:

- `MqttConnection`, con valore "Server=mosquitto;Port=1883",
 - `QoS`, con valore `ExactlyOnce` (equivalente a QoS 2),
 - `Retain`, con valore "true",
 - `CronExpression`, con valore "`* / 5 * * * * ?`", che è la cron expression passata al `QuartzNetTrigger` di `PublisherFunction`.
- un Deployment chiamato `mosquitto`, composto da un Pod che esegue l'immagine `eclipse-mosquitto:2.0`. Eclipse Mosquitto è un broker MQTT open-source. Il container che esegue Mosquitto ha aperta la

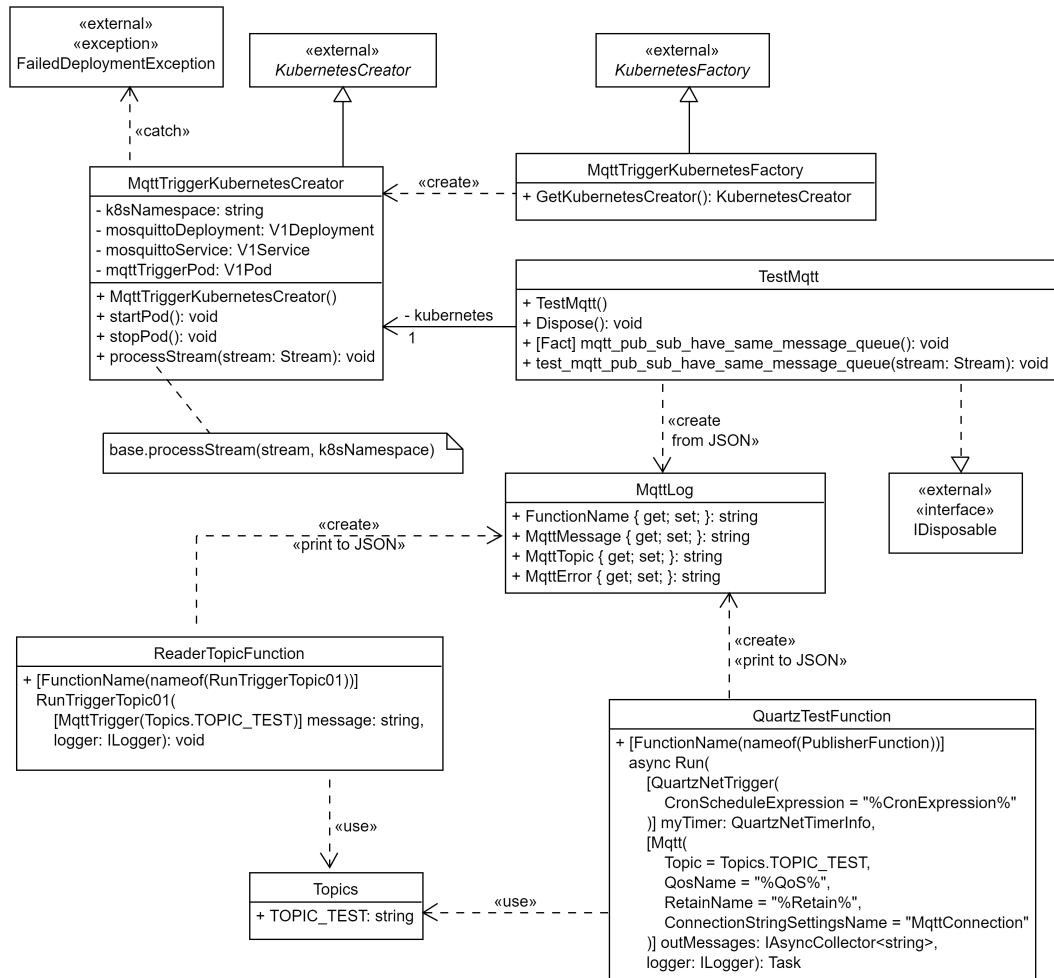


Figura 5.8: Diagramma di classi UML per il test su MQTT

porta TCP 1883 ed esegue il comando

`mosquitto -c /mosquitto-no-auth.conf`, che avvia il broker in una configurazione che permette di collegarsi a esso senza username e password.

- un Service, chiamato anch'esso `mosquitto`, che si collega al Deployment di cui al punto precedente.

I file YAML completi sono consultabili in appendice C.1.

Il test

1. Si creano due code, in una verranno inseriti i messaggi pubblicati, nell'altra quelli ricevuti.
2. si verifica che entrambe le Function siano collegate al broker MQTT. Per ogni riga del log del Pod `mqtttrigger`, si controlla che questa non contenga la sottostringa `"MqttConnection could not connect for"`;
3. riconosciuta una stringa JSON, la si deserializza, si verifica che il campo `MqttError` sia `null` e, in base al campo `FunctionName`, si inserisce la stringa in `MqttMessage` in una delle due code.
4. una volta riconosciute 10 stringhe JSON si confrontano le due code. Sfruttando il fatto che i messaggi MQTT sono pubblicati con QoS 2, se le due code sono uguali il test è superato.

Risultati


Si riportano ora i risultati e i log di esecuzione della pipeline `Beantech.SF.Azure.Functions.Extensions.MQTT.TestingFunction` di Azure DevOps con Build Id 10341, eseguita in data 4 agosto 2021. In figura 5.9 è stato riportato il riepilogo sull'esecuzione della pipeline direttamente dalla UI della piattaforma.

Come illustrato in figura 5.10, entrambi gli stage sono stati eseguiti correttamente e come previsto: il primo ha pubblicato un artifact, il secondo ha eseguito il test dando esito positivo. Stesso discorso per i singoli job, tutti e tre conclusi con esito positivo, come visibile in figura 5.11.

Per quanto riguarda il terzo job, ne era presente un solo test, che si è concluso correttamente, come riportato in figura 5.12.

Analizzando il log di esecuzione del test in appendice C.2, si può notare una curiosa sequenza nell'ordine in cui le stringhe JSON sono stampate: `publish-subscribe-subscribe-publish`, invece che `publish-subscribe` come ci si aspetterebbe. Questo probabilmente è dovuto a due fattori: il primo è il fatto che la Function è asincrona e il log viene stampato solo a invio del messaggio terminato, il secondo è che il QoS utilizzato impone al publisher di attendere finché l'handshake di controllo non è terminato, mentre lo stesso non avviene con il subscriber. Nonostante questo l'ordine di invio e di ricezione dei messaggi è il medesimo per entrambi, infatti se si prendono le stringhe JSON

stampate sul log a coppie (la prima con la seconda, la terza con la quarta e così via), queste condividono il messaggio contenuto in `MqttMessage`.

Manually run by  Riccardo Belliato

[View change](#)









Repository and version	 Beantech.SF.Azure.Functions.Extensions.MQTT  feature  2c7fe142
Time started and elapsed	 Today at 11:36 AM  2m 38s
Related	 0 work items  1 published
Tests and coverage	 100% passed Setup code coverage

Figura 5.9: Riepilogo esecuzione pipeline per MQTT

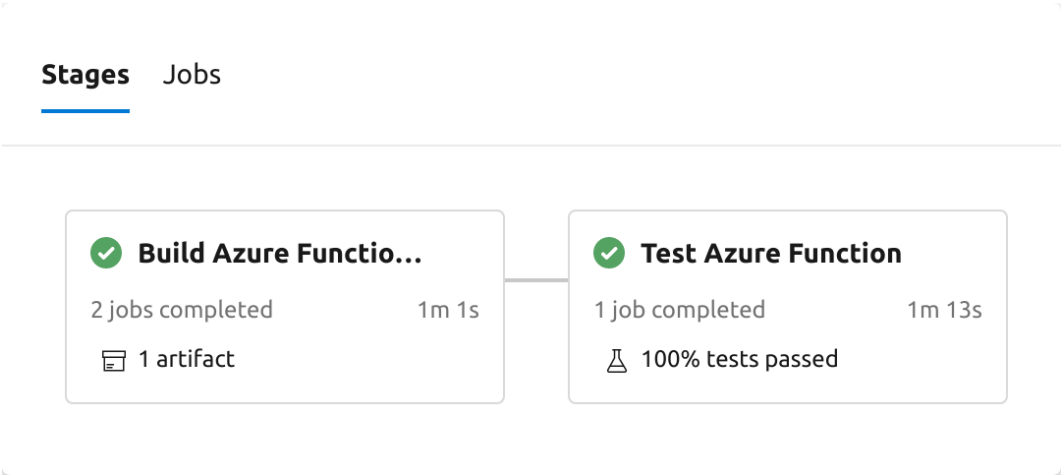


Figura 5.10: Riepilogo esecuzione stages per la pipeline di MQTT

Stages		Jobs
Name		Duration
✓ Build Azure Function as Docker Imager: Build		⌚ 33s
✓ Build Azure Function as Docker Imager: Docker		⌚ 19s
✓ Test Azure Function: Test		⌚ 1m 14s

Figura 5.11: Riepilogo esecuzione job per la pipeline di MQTT

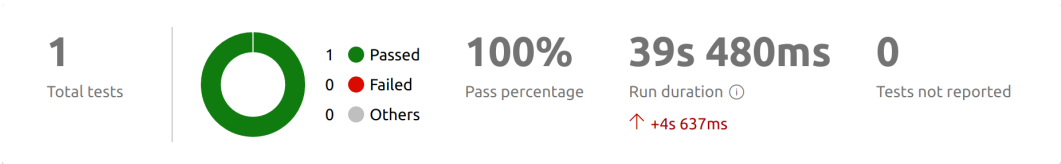


Figura 5.12: Riepilogo test per MQTT

Capitolo 6

Conclusioni e sviluppi futuri

I risultati ottenuti sono in linea con gli obiettivi prefissati nel capitolo 3 in quanto:

- Kubesuite fornisce una base solida e facilmente estendibile per interagire con Kubernetes da codice, non solo a scopo di test;
- la pipeline creata è praticamente già pronta per qualsiasi Azure Function: il Dockerfile rimane lo stesso e, nel caso non sia necessario svolgere test ma solo caricare le immagini Docker su un container registry, è sufficiente estrarre il primo stage in una nuova pipeline cambiando il valore delle variabili. Questo metodo è stato adottato con successo dal team di Brainkin;
- in generale, sfruttando Kubesuite e le convenzioni presentate in questo elaborato, facilita enormemente l'implementazione di test di integrazione su microservizi containerizzati, non solo Azure Functions, e ne riduce il tempo necessario.

Ovviamente tutto il lavoro fin qui presentato ha delle criticità e degli spunti di miglioramento per il futuro:

- i due test presentati sono molto semplici e vanno intesi più a scopo esemplificativo che come test di funzionamento veri e propri,
- Kubesuite non permette di sfruttare appieno le funzionalità messe a disposizione da framework di test avanzati come xUnit.net, come le fixture. Questo è dovuto sia alla natura degli oggetti su cui Kubesuite opera, ma

soprattutto per come è implementata la libreria, che impone a chi la utilizza un workflow stabilito a priori.

- Sempre per il problema di cui al punto precedente, nel caso in cui siano presenti più metodi di test all'interno della stessa classe, l'ambiente di test viene avviato e terminato per ogni metodo di test. Questo in certi contesti (soprattutto per ambienti di test con molti Pod) può rivelarsi inefficiente.
- Da queste considerazioni, un probabile sviluppo futuro è quello di adattare Kubesuite in modo da fornire alle classi di test delle *fixture*¹, invece che delle procedure da eseguire.
- Attualmente è possibile leggere il log solo da un Pod. Può essere interessante fare in modo che sia possibile eseguire test leggendo da due o più log contemporaneamente.
- Kubesuite supporta solo una minima parte degli oggetti k8s disponibili nell'API, tuttavia sfruttando il paradigma del builder per `YamlTypeMap` è molto semplice aggiungere nuovi oggetti supportati (basta creare dei nuovi metodi in `YamlTypeMapBuilder`),
- La pipeline realizzata, allo stato attuale, deve essere copiata per ogni nuovo progetto. Azure DevOps ha la possibilità di creare dei template da importare nelle altre pipeline. Un sicuro sviluppo futuro è quello di convertire la pipeline in un template importabile.

¹le *fixture* sono degli oggetti passati come parametri ai metodi di test che rappresentano l'ambiente o i dati su cui eseguire il test

Appendice A

Codici di esempio ed approfondimenti

A.1 Leggere lo stato di un Pod

Verificare se un Pod è in esecuzione è semplice: come riportato in tabella 1.2, è sufficiente verificare che il campo `Status.Phase` sia uguale a `"Running"`. Per verificare invece se si sono verificati errori in fase di avvio del Pod, bisogna andare a verificare lo stato dei singoli container che lo compongono (campo `Status.ContainerStatuses` del Pod) e, in particolare, il campo `State.Waiting.Reason` dei singoli container. Purtroppo la documentazione ufficiale sui possibili valori di questo campo è praticamente inesistente, per cui quelli riportati provengono dal forum di StackOverflow [14]. I possibili valori di `Reason` che indicano la presenza di errori sono riassunti in tabella A.1.

Stato	Descrizione
ErrImagePull ImagePullBackOff InvalidImageName	Errori in fase di pull dell'immagine Docker dal registry
CrashLoopBackOff CreateContainerConfigError CreateContainerError	Errori in fase di avvio del container

Tabella A.1: Possibili `Reason` di errore per i container in avvio

A.2 Leggere lo stato di Deployment

Verificare che un Deployment sia terminato e sia funzionante è molto diverso rispetto a un Pod. In questo caso bisogna leggere tre campi diversi dall'ultimo elemento della lista che rappresenta lo stato del Deployment (campo `Status.Conditions`), come riportato in sezione 1.2.3. Per verificare lo stato sono rilevanti tre combinazioni di valori dei tre campi da considerare:

- – `Type = "Progressing"`
 - `Status = "False"`
 - `Reason = "ProgressDeadlineExceeded"`

il Deployment ha superato il tempo massimo di avvio a sua disposizione,

- – `Type = "ReplicaFailure"`
 - `Status = "True"`
 - `Reason = "FailedCreate"`

l'avvio di uno o più Pod del Deployment è fallito e non è possibile raggiungere il numero minimo di repliche desiderate,

- – `Type = "Progressing"`
 - `Status = "True"`
 - `Reason = "NewReplicaSetAvailable"`

il Deployment è operativo.

Leggere uno stream riga per riga

Una volta estratto il log dal Pod di test con `processStream` (vedi sezione 4.2.3), questo viene passato come oggetto `Stream` al metodo `testImplementation`.

Per leggere le righe da uno `Stream` in C# si utilizza un oggetto `StreamReader`, il quale ha il compito di convertire i byte dello stream in stringhe di caratteri utilizzando una opportuna codifica (in questo caso, UTF-8), insieme al costrutto `using`.

`using` è un costrutto di C# che viene utilizzato per incapsulare in un blocco di codice l'utilizzo di oggetti di tipo `IDisposable`, come gli `StreamReader`, in modo da avere la certezza che venga invocato `Dispose` o a blocco di codice terminato o se si verifica un'eccezione durante l'esecuzione. Siccome gli

oggetti `IDisposable` solitamente sono astrazioni di file o dispositivi di I/O, il rischio è quello di acquisire queste risorse dal sistema operativo e non rilasciarle mai perchè gli oggetti che le utilizzano non vengono deallocati dal Garbage Collector.

Nel frammento di codice riportato qui sotto è riportato un esempio di utilizzo di `StreamReader` per leggere correttamente da uno stream.

```
using System;
using System.IO;

using (StreamReader sr = new StreamReader(stream))
{
    string line = String.Empty;
    while ((line = sr.ReadLine()) != null)
    {
        // Stampo la riga dello stream sullo standard output
        Console.WriteLine(line);

        //Utilizzo la variabile line...
    }

    /* Quando sr.ReadLine() restituisce null
       le righe dello stream sono terminate.
       A questo punto .NET invoca sr.Dispose().
       Infine sr viene deallocato dal Garbage Collector */

    //Chiudo lo Stream
    stream.Close();
}
```

A.3 Classe di test con Kubesuite

```
1 using Beantech.SF.Azure.Functions.Tests.KubeSuite;
2 using Xunit;
3 using System;
4
5 class KubeTest : IDisposable
6 {
```

```
7     private readonly KubernetesCreator kubernetes;
8
9     //Eseguito prima di ogni test
10    public TestClass()
11    {
12        kubernetes =
13            new KubernetesImplementationFactory().
14                GetKubernetesCreator();
15
16        //Elimina eventuali oggetti pendenti da test precedenti
17        kubernetes.stopPod();
18        try
19        {
20            kubernetes.startPod();
21        }
22        //Ci sono errori nella creazione dell'ambiente
23        catch (FailedDeploymentException e)
24        {
25            kubernetes.stopPod();
26            throw; //Abortisce il test
27        }
28    }
29
30    //Eseguito alla fine di ogni test
31    public void Dispose()
32    {
33        kubernetes.stopPod();
34    }
35
36    [Fact]
37    public void test()
38    {
39        // Tempo limite per il test.
40        var timeLimit = new TimeSpan(...);
41
42        /* Controlla che il test si sia concluso
43        entro il tempo limite, altrimenti è fallito
44        (processStream è true in questo caso) */
45        Assert.False(
46            kubernetes.processStream(testImplementation, timeLimit)
```



```
47         , "Maximum time execution exceeded"
48     );
49 }
50
51 public void testImplementation(Stream stream)
52 {
53     /*
54      Implementazione del test sullo
55      stream del log in arrivo dal pod.
56     */
57 }
58 }
```

A.4 Dockerfile per i test su pipeline con Azure CLI, .NET Core, kubectl e helm

```
1 FROM ubuntu:latest
2
3 ENV DEBIAN_FRONTEND=noninteractive
4 RUN apt-get update \
5     && apt-get upgrade -y \
6     && apt-get install -y --no-install-recommends \
7     wget curl apt-transport-https ca-certificates lsb-release
8     ↪ gnupg
9
10 # Azure-CLI
11 RUN curl -sL https://packages.microsoft.com/keys/microsoft.asc |
12     ↪ gpg --dearmor | tee /etc/apt/trusted.gpg.d/microsoft.gpg >
13     ↪ /dev/null \
14     && AZ_REPO=$(lsb_release -cs) \
15     && echo "deb [arch=amd64]
16     ↪ https://packages.microsoft.com/repos/azure-cli/ $AZ_REPO
17     ↪ main" | tee /etc/apt/sources.list.d/azure-cli.list
18
19 # .NET
```

```

15 RUN wget https://packages.microsoft.com/config/ubuntu/20.04/ \
    ↪ packages-microsoft-prod.deb -O packages-microsoft-prod.deb
    ↪ \
16     && dpkg -i packages-microsoft-prod.deb \
17     && rm *.deb
18
19 RUN apt-get update \
20     && apt-get install -y --no-install-recommends \
21     azure-cli \
22     dotnet-sdk-3.1 \
23     && rm -rf /var/lib/apt/lists/*
24
25 # Helm
26 RUN curl https://raw.githubusercontent.com/helm/helm/master/ \
    ↪ scripts/get-helm-3 |
    ↪ bash
27
28 # Kubectl
29 RUN release=$(curl -L -s https://dl.k8s.io/release/stable.txt) \
30     && curl -LO https://dl.k8s.io/release/$release/bin/linux/ \
    ↪ amd64/kubectl
    ↪ \
31     && install -o root -g root -m 0755 kubectl
    ↪ /usr/local/bin/kubectl \
32     && rm kubectl*

```

A.5 Dockerfile per Azure Functions

```

1 FROM mcr.microsoft.com/azure-functions/dotnet:3.0
2
3 RUN mkdir /app
4
5 ENV AzureWebJobsScriptRoot=/app \
6     AzureFunctionsJobHost__Logging__Console__IsEnabled=true
7
8 ARG PUBLISH_DIR
9
10 COPY ${PUBLISH_DIR} /app

```

A.6 Pipeline YAML di compilazione e test per Azure Functions

```
1 trigger:
2   - none      #Trigger manuale
3
4 pool: 'Beantech-Linux'
5
6 variables:
7   solution:
8     ↪ Beantech.SF.Azure.Functions.Extensions.<nome-estensione>
9   namespace: '$(solution).Func'
10
11 stages:
12   - stage: Build
13     displayName: 'Build Azure Function as Docker Image'
14     variables:
15       artifactName:
16         ↪ <nome-estensione>testingFunction-$(Build.BuildId)
17     jobs:
18       - job: Build
19         container:
20           image: mcr.microsoft.com/dotnet/core/sdk:3.1
21         variables:
22           project: '$(Build.Repository.LocalPath)/$(solution)/」
23             ↪ $(namespace)/$(namespace).csproj'
24         steps:
25           - task: DotNetCoreCLI@2
26             inputs:
27               command: 'restore'
28               projects: '$(project)'
29               feedsToUse: 'select'
30               vstsFeed: '...'
31           - task: DotNetCoreCLI@2
32             inputs:
33               command: 'publish'
```

```

31     publishWebProjects: false
32     projects: '$(project)'
33     arguments: '--configuration Release -o
34         ↪ $(Build.ArtifactStagingDirectory)'
35     zipAfterPublish: false
36     modifyOutputPath: true
37 - bash: |
38     cp $(solution)/$(namespace)/Dockerfile
39     ↪ $(Build.ArtifactStagingDirectory)
40 - publish: $(Build.ArtifactStagingDirectory)
41     artifact: $(artifactName)
42 - deployment: Docker
43     dependsOn: Build
44     variables:
45         repositoryName:
46             ↪ 'beantech.sf.services.function.<nome-estensione>'
47         artifactDir: $(Pipeline.Workspace)/$(artifactName)
48     container:
49         image: 'ubuntu:20.04'
50     environment: 'beantechSFACR'
51     strategy:
52         runOnce:
53             deploy:
54                 steps:
55                     - task: DockerInstaller@0
56                     - task: Docker@2
57                       inputs:
58                         containerRegistry: '...'
59                         command: 'login'
60                     - task: Docker@2
61                       inputs:
62                         containerRegistry: '...'
63                         repository: '$(repositoryName)'
64                         command: 'build'
65                         Dockerfile: '$(artifactDir)/Dockerfile'
66                         tags: 'latest'
67                         arguments: '--build-arg
68                             ↪ PUBLISH_DIR=$(namespace)'
69                     - task: Docker@2
70                       inputs:

```

```

67         containerRegistry: '...'
68         repository: '$(repositoryName)'
69         command: 'push'
70         tags: 'latest'
71
72     - stage: Test
73       displayName: 'Test Azure Function'
74       variables:
75         kubePath: '$(Build.Repository.LocalPath)/$(solution)/」
76           ↳ $(namespace).Tests/Kube'
77         testProject: '$(Build.Repository.LocalPath)/$(solution)/」
78           ↳ $(namespace).Tests/$(namespace).Tests.csproj'
79       dependsOn: Build
80       jobs:
81         - job: Test
82           container:
83             image: 'bkacr002dev.azurecr.io/」
84               ↳ beantech.sf.container.testing:latest'
85             endpoint: '...'
86           steps:
87             - task: AzureCLI@2
88               inputs:
89                 azureSubscription: '...'
90                 scriptType: 'bash'
91                 scriptLocation: 'inlineScript'
92                 inlineScript: 'az aks get-credentials
93                   ↳ --resource-group ... --name ...'
94             #Imposta la variabile d'ambiente KUBE_PATH per tutta
95             ↳ la pipeline
96             #Il comando export su pipeline funziona solo per lo
97             ↳ step corrente
98             - bash: |
99               echo "##vso[task.setvariable
100                 ↳ variable=KUBE_PATH]$(kubePath)"
101             - task: DotNetCoreCLI@2
102               inputs:
103                 command: 'restore'
104                 projects: '$(testProject)'
105                 feedsToUse: 'select'
106                 vstsFeed: '...'

```

```
100     - task: DotNetCoreCLI@2
101       inputs:
102         command: 'test'
103         projects: '$(testProject)'
104         testRunTitle: 'Testing some good stuff'
105
```

Appendice B

Test Quartz.NET

B.1 YAML per test di Quartz.NET

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: quartznet-trigger-test-azfunction
5    labels:
6      name: quartznet-trigger-test-azfunction
7  spec:
8    restartPolicy: OnFailure
9
10  containers:
11  - name: quartznet-trigger-test-azfunction
12    image: bkacr002dev.azurecr.io/
13    ↪ beantech.sf.services.function.quartztrigger:latest
14  env:
15  - name: Occurrences
16    valueFrom:
17      configMapKeyRef:
18        key: Occurrences
19        name: quartznettrigger-test-configmap
20  - name: CronExpression
21    valueFrom:
22      configMapKeyRef:
23        key: CronExpression
```

```

23         name: quartznettrigger-test-configmap
24     resources:
25         limits:
26             memory: "128Mi"
27             cpu: "500m"
28     ports:
29         - containerPort: 80
30 ---
31 apiVersion: v1
32 kind: ConfigMap
33 metadata:
34     name: quartznettrigger-test-configmap
35 data:
36     Occurrences: "10"
37     CronExpression: "*/10 * * * * ?"

```

B.2 Log di test Quartz.NET

```

1 2021-07-28T13:53:33.1936941Z ##[section]Starting: DotNetCoreCLI
2 2021-07-28T13:53:33.1949103Z =====
3 2021-07-28T13:53:33.1949403Z Task           : .NET Core
4 2021-07-28T13:53:33.1949704Z Description  : Build, test,
   ↳ package, or publish a dotnet application, or run a custom
   ↳ dotnet command
5 2021-07-28T13:53:33.1949989Z Version      : 2.187.0
6 2021-07-28T13:53:33.1950201Z Author       : Microsoft
   ↳ Corporation
7 2021-07-28T13:53:33.1950507Z Help         :
   ↳ https://docs.microsoft.com/azure/devops/pipelines/tasks/
   ↳ build/dotnet-core-cli
8 2021-07-28T13:53:33.1950854Z =====
9 2021-07-28T13:53:33.6648612Z [command]/usr/bin/dotnet test
   ↳ /__w/56/s/Beantech.SF.Azure.Functions.Extensions.QuartzNet/
   ↳ Beantech.SF.Azure.Functions.Extensions.QuartzNet.Func.Test/
   ↳ Beantech.SF.Azure.Functions.Extensions.QuartzNet.Func.Test.
   ↳ csproj --logger trx --results-directory
   ↳ /__w/_temp

```



```
10 2021-07-28T13:53:37.8044931Z Test run for
    ↳ /_w/56/s/Beantech.SF.Azure.Functions.Extensions.QuartzNet/
    ↳ Beantech.SF.Azure.Functions.Extensions.QuartzNet.Func.Test/
    ↳ bin/Debug/netcoreapp3.1/Beantech.SF.Azure.Functions.
    ↳ Extensions.QuartzNet.Func.Test.dll(.NETCoreApp,Version=v3.1)
11 2021-07-28T13:53:37.8667253Z Microsoft (R) Test Execution
    ↳ Command Line Tool Version 16.7.1
12 2021-07-28T13:53:37.8671615Z Copyright (c) Microsoft
    ↳ Corporation. All rights reserved.
13 2021-07-28T13:53:37.8671838Z
14 2021-07-28T13:53:37.9459271Z Starting test execution, please
    ↳ wait...
15 2021-07-28T13:53:37.9851346Z
16 2021-07-28T13:53:37.9851964Z A total of 1 test files matched the
    ↳ specified pattern.
17 2021-07-28T13:53:50.5463326Z
    ↳ {"TriggerTime":"2021-07-28T13:53:50.0929978+00:
    ↳ 00","FunctionName":"Beantech.SF.Azure.Functions.Extensions.
    ↳ QuartzNet.Trigger.QuartzNetTimerInfo","Occurence":1}
18 2021-07-28T13:54:00.3782277Z
    ↳ {"TriggerTime":"2021-07-28T13:54:00.0019332+00:
    ↳ 00","FunctionName":"Beantech.SF.Azure.Functions.Extensions.
    ↳ QuartzNet.Trigger.QuartzNetTimerInfo","Occurence":2}
19 2021-07-28T13:54:10.3790862Z
    ↳ {"TriggerTime":"2021-07-28T13:54:10.0007633+00:
    ↳ 00","FunctionName":"Beantech.SF.Azure.Functions.Extensions.
    ↳ QuartzNet.Trigger.QuartzNetTimerInfo","Occurence":3}
20 2021-07-28T13:54:20.3779762Z
    ↳ {"TriggerTime":"2021-07-28T13:54:20.0015726+00:
    ↳ 00","FunctionName":"Beantech.SF.Azure.Functions.Extensions.
    ↳ QuartzNet.Trigger.QuartzNetTimerInfo","Occurence":4}
21 2021-07-28T13:54:30.3781915Z
    ↳ {"TriggerTime":"2021-07-28T13:54:30.0010672+00:
    ↳ 00","FunctionName":"Beantech.SF.Azure.Functions.Extensions.
    ↳ QuartzNet.Trigger.QuartzNetTimerInfo","Occurence":5}
22 2021-07-28T13:54:40.3794280Z
    ↳ {"TriggerTime":"2021-07-28T13:54:40.0009354+00:
    ↳ 00","FunctionName":"Beantech.SF.Azure.Functions.Extensions.
    ↳ QuartzNet.Trigger.QuartzNetTimerInfo","Occurence":6}
```

```

23 2021-07-28T13:54:50.3869493Z
    ↳ {"TriggerTime":"2021-07-28T13:54:50.0007985+00:
    ↳ 00","FunctionName":"Beantech.SF.Azure.Functions.Extensions.
    ↳ QuartzNet.Trigger.QuartzNetTimerInfo","Occurence":7}
24 2021-07-28T13:55:00.3788860Z
    ↳ {"TriggerTime":"2021-07-28T13:55:00.0009757+00:
    ↳ 00","FunctionName":"Beantech.SF.Azure.Functions.Extensions.
    ↳ QuartzNet.Trigger.QuartzNetTimerInfo","Occurence":8}
25 2021-07-28T13:55:10.3777781Z
    ↳ {"TriggerTime":"2021-07-28T13:55:10.0006832+00:
    ↳ 00","FunctionName":"Beantech.SF.Azure.Functions.Extensions.
    ↳ QuartzNet.Trigger.QuartzNetTimerInfo","Occurence":9}
26 2021-07-28T13:55:20.3784498Z
    ↳ {"TriggerTime":"2021-07-28T13:55:20.0008896+00:
    ↳ 00","FunctionName":"Beantech.SF.Azure.Functions.Extensions.
    ↳ QuartzNet.Trigger.QuartzNetTimerInfo","Occurence":10}
27 2021-07-28T13:55:29.6614107Z Results File:
    ↳ /_w/_temp/_8045d38871c1_2021-07-28_13_55_29.trx
28 2021-07-28T13:55:29.6616101Z
29 2021-07-28T13:55:29.6629440Z Test Run Successful.
30 2021-07-28T13:55:29.6632155Z Total tests: 1
31 2021-07-28T13:55:29.6635379Z Passed: 1
32 2021-07-28T13:55:29.6639691Z Total time: 1.8611 Minutes
33 2021-07-28T13:55:30.9888933Z Result Attachments will be stored
    ↳ in LogStore
34 2021-07-28T13:55:31.0280694Z Run Attachments will be stored in
    ↳ LogStore
35 2021-07-28T13:55:31.1200071Z ##[section]Async Command Start:
    ↳ Publish test results
36 2021-07-28T13:55:31.5024942Z Publishing test results to test run
    ↳ '1016134'.
37 2021-07-28T13:55:31.5086160Z TestResults To Publish 1, Test run
    ↳ id:1016134
38 2021-07-28T13:55:31.5149794Z Test results publishing 1,
    ↳ remaining: 0. Test run id: 1016134
39 2021-07-28T13:55:32.6930529Z Published Test Run :
    ↳ https://dev.azure.com/beantech/Beantech.SF/_TestManagement/
    ↳ Runs?runId=1016134&_a=runCharts
40 2021-07-28T13:55:32.8357356Z ##[section]Async Command End:
    ↳ Publish test results

```

41 2021-07-28T13:55:32.8359553Z ##[section]Finishing: DotNetCoreCLI

Appendice C

Test MQTT

C.1 YAML per test di MQTT

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: mosquitto
5  spec:
6    # 1 minuto per essere operativo
7    progressDeadlineSeconds: 60
8    selector:
9      matchLabels:
10       app: mosquitto
11    template:
12      metadata:
13       labels:
14         app: mosquitto
15      spec:
16       containers:
17       - name: mosquitto
18         image: eclipse-mosquitto:2.0
19         resources:
20           limits:
21             memory: "128Mi"
22             cpu: "500m"
23       ports:
```

```

24         - containerPort: 1883
25         command: ["mosquitto", "-c", "/mosquitto-no-auth.conf"]
26     ---
27     apiVersion: v1
28     kind: Service
29     metadata:
30       name: mosquitto
31     spec:
32       selector:
33         app: mosquitto
34       ports:
35         - port: 1883
36           targetPort: 1883

```

```

1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: mqtttrigger
5    labels:
6      name: mqtttrigger
7  spec:
8    restartPolicy: Always
9    containers:
10     - name: mqtttrigger
11       image: bkacr002dev.azurecr.io/
12         ↪ beantech.sf.services.function.mqtttrigger:latest
13     resources:
14       limits:
15         memory: "128Mi"
16         cpu: "500m"
17     env:
18       - name: MqttConnection
19         value: "Server=mosquitto;Port=1883"
20       - name: QoS
21         value: AtLeastOnce
22       - name: Retain
23         value: "true"
24       - name: CronExpression
25         value: "*/5 * * * * ?"

```

```

25     ports:
26         - containerPort: 80

```

C.2 Log di test MQTT

```

1  2021-08-04T09:37:58.1918511Z ##[section]Starting: DotNetCoreCLI
2  2021-08-04T09:37:58.1928856Z =====
3  2021-08-04T09:37:58.1929153Z Task           : .NET Core
4  2021-08-04T09:37:58.1929458Z Description  : Build, test,
   ↪ package, or publish a dotnet application, or run a custom
   ↪ dotnet command
5  2021-08-04T09:37:58.1929752Z Version      : 2.187.0
6  2021-08-04T09:37:58.1929964Z Author       : Microsoft
   ↪ Corporation
7  2021-08-04T09:37:58.1930266Z Help         :
   ↪ https://docs.microsoft.com/azure/devops/pipelines/tasks/
   ↪ build/dotnet-core-cli
8  2021-08-04T09:37:58.1930618Z =====
9  2021-08-04T09:37:58.6991466Z [command]/usr/bin/dotnet test
   ↪ /__w/54/s/Beantech.SF.Azure.Functions.Extensions.MQTT/
   ↪ Beantech.SF.Azure.Functions.Extensions.MQTT.Func.Tests/
   ↪ Beantech.SF.Azure.Functions.Extensions.MQTT.Func.Tests.
   ↪ csproj --logger trx --results-directory
   ↪ /__w/_temp
10 2021-08-04T09:38:03.6442537Z Test run for
   ↪ /__w/54/s/Beantech.SF.Azure.Functions.Extensions.MQTT/
   ↪ Beantech.SF.Azure.Functions.Extensions.MQTT.Func.Tests/bin/
   ↪ Debug/netcoreapp3.1/Beantech.SF.Azure.Functions.Extensions.
   ↪ MQTT.Func.Tests.dll(.NETCoreApp,Version=v3.1)
11 2021-08-04T09:38:03.7149594Z Microsoft (R) Test Execution
   ↪ Command Line Tool Version 16.7.1
12 2021-08-04T09:38:03.7151105Z Copyright (c) Microsoft
   ↪ Corporation. All rights reserved.
13 2021-08-04T09:38:03.7151733Z
14 2021-08-04T09:38:03.8118793Z Starting test execution, please
   ↪ wait...
15 2021-08-04T09:38:03.8924153Z

```

```
16 2021-08-04T09:38:03.8925671Z A total of 1 test files matched the
    ↳ specified pattern.
17 2021-08-04T09:38:21.1760844Z
    ↳ {"FunctionName":"publish","MqttMessage":"Executing MQTT at
    ↳ 2021-08-04 09:38:15","MqttTopic":"test","MqttError":null}
18 2021-08-04T09:38:21.2761429Z
    ↳ {"FunctionName":"subscribe","MqttMessage":"Executing MQTT at
    ↳ 2021-08-04 09:38:15","MqttTopic":"test","MqttError":null}
19 2021-08-04T09:38:25.9183157Z
    ↳ {"FunctionName":"subscribe","MqttMessage":"Executing MQTT at
    ↳ 2021-08-04 09:38:20","MqttTopic":"test","MqttError":null}
20 2021-08-04T09:38:25.9237500Z
    ↳ {"FunctionName":"publish","MqttMessage":"Executing MQTT at
    ↳ 2021-08-04 09:38:20","MqttTopic":"test","MqttError":null}
21 2021-08-04T09:38:30.9151834Z
    ↳ {"FunctionName":"publish","MqttMessage":"Executing MQTT at
    ↳ 2021-08-04 09:38:25","MqttTopic":"test","MqttError":null}
22 2021-08-04T09:38:30.9154035Z
    ↳ {"FunctionName":"subscribe","MqttMessage":"Executing MQTT at
    ↳ 2021-08-04 09:38:25","MqttTopic":"test","MqttError":null}
23 2021-08-04T09:38:35.9137197Z
    ↳ {"FunctionName":"subscribe","MqttMessage":"Executing MQTT at
    ↳ 2021-08-04 09:38:30","MqttTopic":"test","MqttError":null}
24 2021-08-04T09:38:35.9138505Z
    ↳ {"FunctionName":"publish","MqttMessage":"Executing MQTT at
    ↳ 2021-08-04 09:38:30","MqttTopic":"test","MqttError":null}
25 2021-08-04T09:38:40.9157762Z
    ↳ {"FunctionName":"publish","MqttMessage":"Executing MQTT at
    ↳ 2021-08-04 09:38:35","MqttTopic":"test","MqttError":null}
26 2021-08-04T09:38:40.9197116Z
    ↳ {"FunctionName":"subscribe","MqttMessage":"Executing MQTT at
    ↳ 2021-08-04 09:38:35","MqttTopic":"test","MqttError":null}
27 2021-08-04T09:38:43.3656587Z Results File:
    ↳ /__w/_temp/_46882c540453_2021-08-04_09_38_43.trx
28 2021-08-04T09:38:43.3659194Z
29 2021-08-04T09:38:43.3667602Z Test Run Successful.
30 2021-08-04T09:38:43.3669977Z Total tests: 1
31 2021-08-04T09:38:43.3670469Z Passed: 1
32 2021-08-04T09:38:43.3672610Z Total time: 39.4572 Seconds
33 2021-08-04T09:38:45.1952706Z Result Attachments will be stored
    ↳ in LogStore
```

```
34 2021-08-04T09:38:45.2359291Z Run Attachments will be stored in
    ↳ LogStore
35 2021-08-04T09:38:45.3183508Z ##[section]Async Command Start:
    ↳ Publish test results
36 2021-08-04T09:38:45.7298660Z Publishing test results to test run
    ↳ '1016506'.
37 2021-08-04T09:38:45.7358354Z TestResults To Publish 1, Test run
    ↳ id:1016506
38 2021-08-04T09:38:45.7421184Z Test results publishing 1,
    ↳ remaining: 0. Test run id: 1016506
39 2021-08-04T09:38:46.8132905Z Published Test Run :
    ↳ https://dev.azure.com/beantech/Beantech.SF/_TestManagement/_
    ↳ Runs?runId=1016506&_a=runCharts
40 2021-08-04T09:38:46.9563316Z ##[section]Async Command End:
    ↳ Publish test results
41 2021-08-04T09:38:46.9566600Z ##[section]Finishing: DotNetCoreCLI
```

Bibliografia

- [1] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.
- [2] Katrina Clokier. A practical guide to testing in devops, 2017.
- [3] Kubernetes Documentation. Configmaps — kubernetes. <https://kubernetes.io/docs/concepts/configuration/configmap/>.
- [4] Kubernetes Documentation. Deployments — kubernetes. <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>.
- [5] Kubernetes Documentation. I componenti di kubernetes. <https://kubernetes.io/it/docs/concepts/overview/components/s>.
- [6] Kubernetes Documentation. Namespaces — kubernetes. <https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/>.
- [7] Kubernetes Documentation. Pod lifecycle — kubernetes. <https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/>.
- [8] Kubernetes Documentation. Pods — kubernetes. <https://kubernetes.io/docs/concepts/workloads/pods/>.
- [9] Kubernetes Documentation. Secrets — kubernetes. <https://kubernetes.io/docs/concepts/configuration/secret/>.
- [10] Kubernetes Documentation. Service — kubernetes. <https://kubernetes.io/docs/concepts/services-networking/service/>.

- [11] Microsoft Documentation. What is azure pipelines? <https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops>, 2021.
- [12] Red Hat. Cos'è l'orchestrazione dei container? <https://www.redhat.com/it/topics/containers/what-is-container-orchestration>.
- [13] Docker Inc. What is a container? <https://www.docker.com/resources/what-container>.
- [14] Ines. List of all reasons for container states in kubernetes. <https://stackoverflow.com/questions/57821723/list-of-all-reasons-for-container-states-in-kubernetes>.
- [15] Introduzione a json. <https://www.json.org/json-it.html>.
- [16] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Comput. Surv.*, 52(6), November 2019.
- [17] Hongwei Li, Junsheng Wang, Hua Dai, and Bang Lv. Research on microservice application testing system. In *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, pages 363–368, 2020.
- [18] Ruth W. Macarthy and Julian M. Bass. An empirical taxonomy of devops in practice. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 221–228, 2020.
- [19] Garrett McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410, 2017.
- [20] Microsoft. Che cos'è il cloud computing ibrido. <https://azure.microsoft.com/it-it/overview/what-is-hybrid-cloud-computing/>.
- [21] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

-
- [22] Muhammad Waseem, Peng Liang, Gastón Márquez, and Amleto Di Salle. Testing microservices architecture-based applications: A systematic mapping study. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, pages 119–128, 2020.