Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica per il Management

Design e Sviluppo di un Framework di Testing per Microservizi

Relatore:
Chiar.mo Prof.
DAVIDE SANGIORGI

Presentata da:
MATTEO SANFELICI

Correlatore:
Dott.
SAVERIO GIALLORENZO

Sessione 2
Anno Accademico 2016-2017

Indice

Parte I

Introduzione

Capitolo 1

Obbiettivo della Tesi

Un'introduzione generale è d'obbligo per il progetto che verrà descritto con questa tesi, *Design e Sviluppo di un Framework di testing per microservizi.*

La prima parte di Introduzione va ad introdurre il contesto della programmazione di sistemi distribuiti, i problemi legati ad essa e come le architetture a microservizi possano aiutare nello sviluppo e nella manutenzione di tali sistemi. In seguito viene presentato il testing di sistemi distribuiti, le problematiche legate ad esso e i principi del Development Operation e del concetto di framework di testing. Questa è la parte in cui si parla di ciò che ha portato alla progettazione e allo sviluppo dei temi trattati in questa tesi, ma anche dei concetti che stanno alla base della tesi nella sua interezza.

Assodata la necessità di un tool a supporto del testing di architetture distribuite e di architetture a microservizio, in particolare, la seconda parte si concentra sul design di un possibile framework di testing per sistemi a microservizi. Tra le componenti del framework proposto, il primo componente fondamentale è rappresentato dallo Unit Testing Automatizzato. Questa tesi si focalizza sullo sviluppo di questo primo, fondamentale componente, di cui, come conclusione della seconda parte, viene spiegata la progettazione.

La terza e ultima parte si occupa di scendere nello specifico dello sviluppo di ciò che è stato spiegato nella parte di Design. Nella parte trovano posto le scelte implementative per il framework di Unit Testing, come la scelta del linguaggio di sviluppo, le scelte di tipo architetturale e quelle che riguardano l'interazione con gli utenti (e.g. come scrivere ed eseguire il test di un microservizio). Successivamente si spiegherà come si è riusciti ad automatizzare lo Unit Test e verranno descritti gli strumenti utilizzati

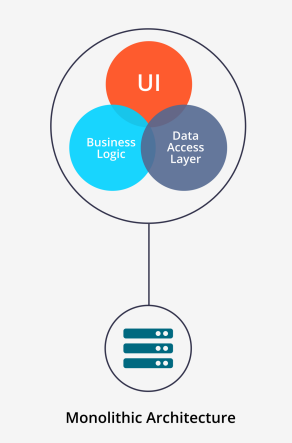
La quarta parte, che conclude la tesi, elenca i contributi apportati e possibili lavori futuri.

Capitolo 2

Testing di Architetture a Microservizi

2.1 Sviluppo di sistemi distribuiti e architetture a microservizi

Un sistema distribuito è una rete composta da computer autonomi che sono connessi tra loro, i quali comunicano e coordinano le proprie operazioni passandosi messaggi.
Un'architettura a microservizi è un vero e proprio sistema distribuito, dove i vari componenti (servizi) interagiscono tra loro per portare a termine un compito comune.
I sistemi distribuiti basati su architetture a microservizi stanno da tempo sostituendo quelli basati su un architetture monolitiche.



Sistemi Monolitici In questa architettura lo sviluppo software prevede lo sviluppo di una sola base di codice compilato e distribuito all'interno di un **unico pacchetto**. Questo tipo di sviluppo considera l'applicazione come

una singola entità. L'approccio è legato al mondo classico di sviluppare applicazioni in cui l'esecuzione del programma viene intesa all'interno di un unico elaboratore. L'approccio monolitico è opportuno per progetti di dimensioni ridotte poco soggetti a modifiche.

Al contrario, per applicazioni complesse e di grosse dimensioni l'approccio monolitico presenta limitazioni riguardo l'evoluzione dell'applicazione e la sua manutenzione; e.g. ogni funzionalità aggiunta implica possibili problemi di compatibilità con le componenti preesistenti ed ogni aggiornamento necessita lo spegnimento dell'applicativo corrente e l'installazione del nuovo. Inoltre l'unico modo per poter scalare un applicazione monolitica è la duplicazione, che spreca risorse, duplicando tutte le funzionalità dell'applicazione interessata, mentre è probabile che solo alcune sottoparti siano effettivamente interessate da un incremento di richieste da parte degli utenti.

Multi-tier Per risolvere alcuni degli svantaggi delle applicazioni monolitiche, si è pensato prima ad una scomposizione del software monolitico sul piano logico, pensando una applicazione come una serie di strati che collaborano in modo gerarchico.

L'esempio tipico di tale divisione è lo **strato di logica** di gestione di un'applicazione e lo **strato presentazionale** di grafica. Se ad esempio è necessario effettuare computazioni intensive dal lato dello strato gestionale, grazie alla loro separazione è possibile scalare solo quello strato, mantenendo inalterato quello presentazionale.

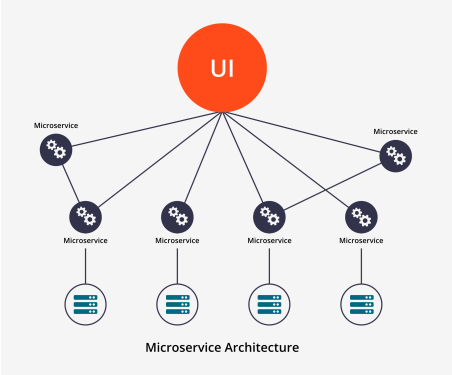
Service Oriented Applications (SOA) Il passo successivo è stato scomporre lo strato logico in funzionalità indipendenti, creando un insieme di **Servizi** indipendenti tra loro, perciò facilmente scalabili e più facili da gestire.

Benché le SOA promettessero un modello di sviluppo e gestione più agile e adatto ai sistemi distribuiti, nella pratica il modello architetturale venne aggravato da astrazioni che favorivano la creazione di monoliti, ai quali si aggiungevano i costi delle tecnolo-

gie di comunicazione SOA (e.g., bus di comunicazione, descrittori di interfacce, etc.), diminuendo la flessibilità del sistema.

2.2 Microservizi e sistemi distribuiti

L'ultima iterazione sul concetto di servizio e architettura di software distribuiti è il modello architetturale dei microservizi. I microservizi vanno a raffinare il paradigma SOA scomponendo ogni funzionalità di business in un servizio completamente autonomo, ma capace di collaborare con altri microservizi.



Al contrario di SOA, i microservizi non impongono pesanti strati di astrazione, ma prediligono l'utilizzo di tecnologie di comunicazione aperte e semplici, come HTTP e JSON.

2.3 Vantaggi e Svantaggi delle architetture a microservizi

2.3.1 Vantaggi

Coi microservizi si favorisce la scalabilità e alcune operazioni come mantenimento, correzione di bug e aggiornamento del sistema diventano molto più semplici. I principali vantaggi riscontrabili sono:

Sistema meno fragile (Resilienza) Essendo ogni microservizio autonomo e isolato, in caso di crash di un microservizio, tale problema non si ripercuoterà su tutto il sistema,

che continuerà a funzionare. Il punto di crash sarà facilmente isolato, corretto e rimesso in funzione senza la necessità di riavviare l'intero sistema.

Interazioni e Deploy più veloci Si produrrà codice semplice da capire e gli sviluppatori potranno concentrarsi su un'unica funzionalità. I possibili aggiornamenti riguarderanno un servizio nello specifico e quindi il deploy di un aggiornamento relativo sarà molto più veloce e facile favorendo processi di sviluppo quali il Continuous Delivery[1]).

Scalabilità Scalare un singolo servizio è molto semplice e può essere fatto in maniera ottimizzata rispetto al carico sul singolo microservizio, anziché scalare un intero sistema monolitico. Ogni servizio può funzionare sull'hardware che più si adatta al suo consumo di risorse ottimizzando prestazioni e costi.

Flessibilità della tecnologia Si ottiene fornendo un'Interfaccia di Programmazione dell'Applicazione (API) per comunicare con un microservizio. Tale impostazione permette di non essere vincolati a una tecnologia e, lasciando scegliere di volta in volta la migliore per un certo scopo, vengono astratti i dettagli di implementazione delle sue funzionalità, fruibili in maniera standardizzata da parte di altri servizi, sviluppati con tecnologie differenti e possibilmente incompatibili tra loro.

Al contrario, in architetture monolitiche la scelta del linguaggio va a limitare lo sviluppo di ogni sua componente e rende difficile il passaggio a tecnologie differenti.

Riuso e Componibilità La modularità di un microservizio favorisce l'eventuale riuso in applicazioni future o la componibilità tra porzioni di applicazioni diverse ottimizzando costi e velocità di sviluppo.

[1]https://en.wikipedia.org/wiki/Continuous_delivery

2.3.2 Svantaggi

Come ogni soluzione, anche i microservizi pongono alcune limitazioni e svantaggi da considerare in sede di scelta dell'architettura di un'applicazione distribuita.

Comunicazione tra microservizi Il basso accoppiamento e la comunicazione su rete possono portare a problemi come la latenza e potenziali fallimenti della comunicazione che rallentano l'intero flusso dei dati e peggiorano l'esperienza d'uso del sistema da parte dell'utente. Generalmente per rendere affidabile la comunicazione tramite la rete viene aggiunta una coda per gestire i messaggi, che implica un ulteriore componente da considerare e gestire all'interno dell'architettura.

Coerenza dei dati In ogni sistema distribuito è un problema saper gestire in modo adeguato la persistenza dei dati in database replicati (e.g., per evitare possibili incongruenze) e in transito tra i microservizi.

Test La mancanza di veri proprio framework di testing di applicazioni distribuite testimonia la difficoltà di tale pratica. L'alto grado di distribuzione dei microservizi incrementa tale difficoltà, sopratutto se considerata nel contesto più ampio del testing di intere architetture, oltre che dei singoli microservizi.

2.4 Microservizi: presente e futuro

L'approccio a microservizi sta prendendo sempre più piede perché i vantaggi che offre nel campo dello sviluppo di sistemi distribuiti sopperiscono agli svantaggi e alle loro problematiche tutt'ora irrisolte.

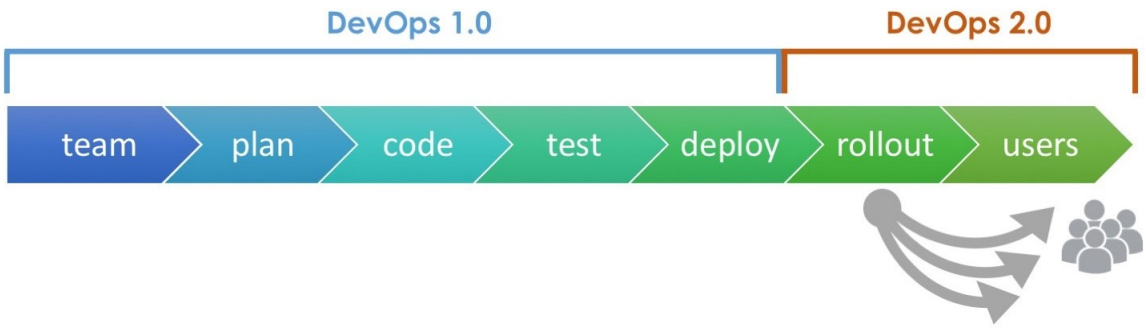
Il contributo del lavoro di questa tesi mira a fornire una possibile soluzione ad uno dei principali problemi dei microservizi: il testing. Nella pratica, il presente lavoro contiene due contributi; il design di un framework per il testing di microservizi e l'implementa-

zione di un core di tale framework.

Per procedere col primo contributo di design del framework, è stato studiato il metodo di sviluppo del Development Operations (DevOps), prendendo da esso le specifiche, i requisiti e i desiderata di un framework di testing che segue tali principi.

Capitolo 3

DevOps: Development Operation



3.1 I principi del DevOps

3.1.1 DevOps 1.0

Lo sviluppo di un sistema software sta diventando sempre più complesso, dove team di sviluppatori e amministratori di sistema separati sviluppano componenti in parallelo.

Anche in progetti relativamente piccoli si hanno 2 (o 3) team: (I) **frontend**, sviluppo della parte di un sistema software che gestisce l'interazione con l'utente o con sistemi esterni che producono dati di ingresso (II) **backend**, sviluppo della parte che elabora i dati generati dal front end. A volte si può trovare anche un terzo team per (III) **gestione database**, il quale si occupa di ottimizzare i dati salvati e l'interazione con essi da parte

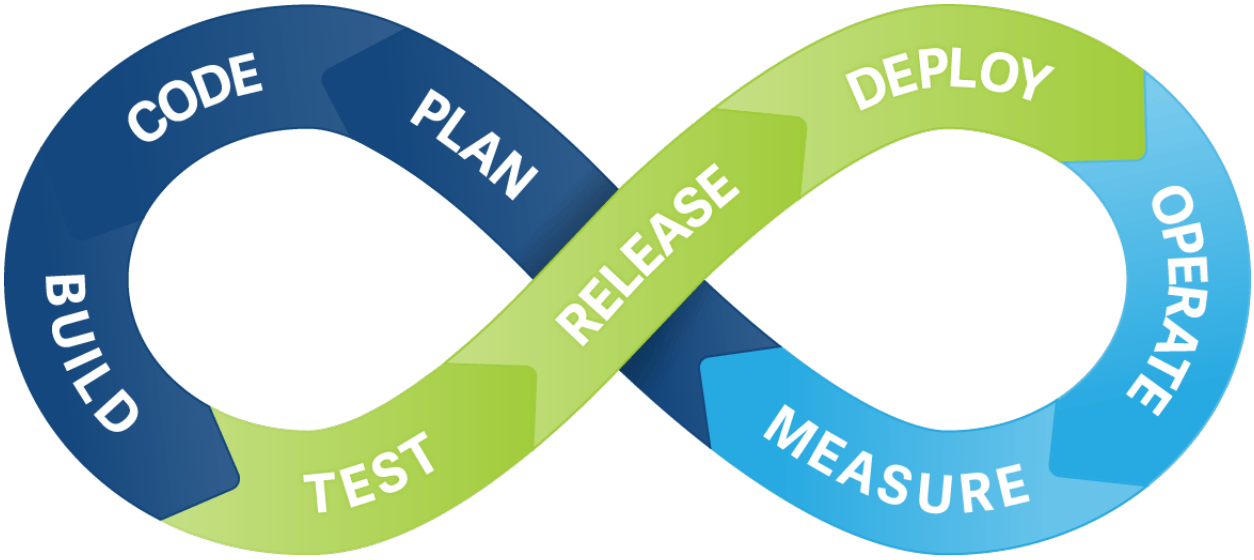
del backend.

DevOps si occupa di stabilire una serie di processi per automatizzare e coordinare l'interazione tra sviluppatori e amministratori di sistema.

3.1.2 DevOps 2.0

Oltre ai team tecnici, si possono trovare team non tecnici quali marketing, design o business. Negli ultimi anni DevOps2.0 sta cercando di portare i processi del DevOps anche al lato non tecnico del software, offrendo interfacce per gestire release di nuove funzionalità in completa sicurezza senza la necessità di andare ad interagire con i team tecnici.

3.2 Continuous Integration e Continuous Deployment



3.2.1 Continuous Integration

Il Continuous Integration (CI) è una pratica software che richiede che gli sviluppatori "integrino" costantemente il software aggiornato, prodotto anche a scadenze intragiornaliere. Da specifica, ogni aggiunta viene verificata (ed eventualmente compilata) da un processo automatizzato, in modo tale che i team siano subito in grado di scovare errori.

Integrando spesso si può trovare e risolvere errori più frequentemente, risolvendo eventuali problemi al momento dell'integrazione di ogni piccolo aggiornamento. Ciò diverge sensibilmente dal modello tradizionale dove i moduli (e.g., servizi) vengono sviluppati indipendentemente e integrati solo al momento di una milestone. Così facendo, durante lo sviluppo i moduli possono accumulare errori che si manifestano solo al momento dell'integrazione, la cui correzione richiede una costosa operazione di analisi dell'intero codice dell'applicativo aggiornato.

I principali benefici del continuous integration sono:

- integrazione più semplice e breve, dando la possibilità di consegnare software aggiornato più rapidamente.
- eliminare eventuali problemi di integrazione al momento dell'aggiornamento.
- ridurre sensibilmente i tempi di debugging.

3.2.2 Continuous Deployment

Passate tutte le fasi di integrazione un software è pronto per essere messo in funzione. La fase di messa in esecuzione di un software viene chiamata deployment.

Prima della definizione e dell'applicazione del Continuous Integration il deployment del software consisteva nel dover integrare manualmente e compilare tutto il sistema aggiornato, era necessario rimuovere il sistema precedente e caricare quello nuovo (rendendo il servizio inagibile per gli utenti).

Includendo il deployment all'interno del processo di Continuous Integration risulta molto più semplice aggiornare e mettere in esecuzione il codice degli applicativi. Il continuous deployment (CD) può essere visto come un'estensione del continuous integration, che punta a minimizzare il tempo che passa tra la scrittura del codice di una nuova funzionalità e il suo utilizzo effettivo da parte dell'utente.

Per raggiungere questo obbiettivo si fa affidamento su una struttura automatizzata in modo tale che, dopo aver passato tutte le fasi di integration, venga resa operativa la nuova funzionalità e che, in caso di errore, si torni a eventuali versioni precedenti, minimizzando i problemi per gli utenti.

3.3 Conclusioni

I principi del DevOps sono indispensabili per tenere il passo di un mondo in continua evoluzione. Tramite processi automatizzati di integrazione e deployment si velocizza di molto sia la parte di integrazione e di testing, sia la parte di deployment.

Nel prossimo capitolo viene approfondita la pratica del testing secondo i principi del DevOps e le difficoltà nell'automatizzarlo in sistemi distribuiti.

Capitolo 4

Testing

Il testing non è altro che una fase del processo di continuous integration, detta anche fase di validazione, che deve analizzare il software e controllare che tutto funzioni come ci si aspetta.

4.1 Perché il testing è importante

Il software testing è importante perché spesso vengono fatti errori, alcuni non importanti e che non vanno a inficiare il funzionamento del software, mentre altri sono estremamente costosi e pericolosi.

Mentre si scrive codice viene sempre riletto ciò che è appena stato prodotto e viene controllato che tutto sia scritto correttamente. Purtroppo alcuni errori derivano da assunzioni errate o punti ciechi, quindi potrebbero sfuggire a questo primo controllo.

Per questo esistono software per gestire serie di test su singole funzioni o su interi sistemi. I motivi per ritenere importanti il testing sono molteplici, i principali sono:

- scovare più difetti e errori possibili, fatti durante lo sviluppo del software

- se fatto dall'inizio della produzione del software, riduce i costi della correzione di un errore dopo mesi di sviluppo
- serve a verificare che anche sotto stress il sistema abbia performance ottimali e stabili
- serve a sostenere con dati oggettivi la qualità dell'applicazione prodotta

4.2 Perché il testing è così difficile

4.2.1 Difficoltà generali e automatizzazione

Testare un software non è mai cosa banale, un buon software di testing dovrebbe testare ogni funzione, modulo, componente con tutti gli input possibili che può ricevere e in tutte le situazioni concepibili in cui può trovarsi per avere un'accuratezza (o copertura) del 100%, cosa generalmente impossibile.

Inoltre ogni test eseguito dal software di testing non è mai completamente automatizzato, ma richiede l'interazione di uno sviluppatore che lo configura, dando adito ad ulteriori errori.

4.2.2 Testare sistemi a microservizi

In un paradigma a microservizi il sistema da testare è distribuito. Ogni microservizio si occupa di un piccolo numero di operazioni e può essere implementato in qualsiasi linguaggio.

Un software di testing in ambienti distribuiti a microservizi, oltre ai normali problemi quali copertura del test e complessità, incorre quindi in altri problemi quali:

- verifica di codice in linguaggio potenzialmente diverso da un microservizio all'altro

- necessità di validare un sistema distribuito cercando di stressare certi nodi piuttosto che altri
- trovare un modo per controllare come si comporta un sistema distribuito in caso di failure.
- se un microservizio viene scalato, verifica che i dati vengano propagati a tutto il cluster di istanze di quel microservizio.

4.3 Related works

4.3.1 Netflix e Simian Army

Il problema di testare un sistema distribuito a microservizi è molto sentito al giorno d'oggi, per esempio Netflix ha ideato un vero e proprio framework chiamato Simian Army, composto da tool che, tra le altre cose, si occupano di iniettare errori e malfunzionamenti nel sistema e verificare che tutto continui a funzionare (Chaos Monkey[1]) oppure verificare che ogni istanza di servizi disponibili si stia comportando come prefissato (Conformity Monkey[2]).

[1]https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey
[2]https://github.com/Netflix/SimianArmy/wiki/Conformity-Home

4.4 Conclusioni

Lo sviluppo di sistemi distribuiti e in particolare quello delle architetture a microservizi necessita di una serie di tool per testare il corretto funzionamento del software in modo automatico.

Nel prossimo capitolo verrà esposto il design di un possibile framework di test per microservizi.

Parte II

Design di un Framework di Testing per Microservizi

Capitolo 5

Design Roadmap

Il presente capitolo contiene il design di un possibile framework di testing per architetture a microservizi. Tale design comprende due artefatti principali: la definizione di una roadmap dei tool utili al testing dei microservizi, dai singoli applicativi a intere architetture fino a strumenti integrati col deployment.

Come linea guida per lo sviluppo di questa roadmap è stato seguito il testo "The DevOps 2.0 Toolkit (Automating the Continuous Deployment Pipeline with Containerized Microservices)"[1].

[1]Autore: Victor Farcic. Autore anche di "Test-Driven Java Development" ed esperto di Microservizi, Continuous Deployment and Test-Driven Development (TDD). Ora Senior Consultant presso CloudBees azienda che sviluppa Jenkins e offre supporto ad altre aziende nell'utilizzo di un processo di sviluppo di Continuous Delivery (Continuous Integration e Deployment automatizzati).

La roadmap ideata comprende una serie di tool che automatizzano il processo di testing su vari livelli, fino ad arrivare al deployment. Gli step principali possono essere riassunti in 5 passaggi: Static Testing, Unit Testing, Integration Testing, System (Integration) Testing, Deployment Hooks.

5.1 Static Testing

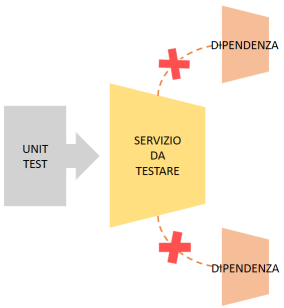
È una tecnica di test nella quale il software viene testato senza eseguire il codice. Può essere divisa in (I) Revisione e (II) Analisi Statica.

Revisione, tipicamente usata per trovare ed eliminare errori o imprecisioni nel codice o nelle documentazioni; si differenzia per formalità (informale, passo per passo, peer review, ispezione).

Analisi Statica, dove il codice viene analizzato da tool per scovare difetti strutturali che potrebbero dare problemi o per verificare che il codice segua le regole stilistiche imposte

5.2 Unit Testing

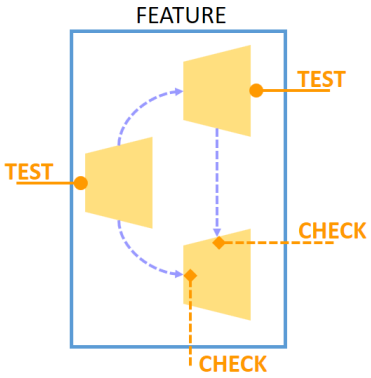
È una tecnica di test fatta dallo stesso sviluppatore che ha scritto un *singolo modulo* per verificare che non ci siano problemi. Si concentra sulla correttezza funzionale del modulo *isolato* dal resto del sistema (dipendenze).
Porta ad alcuni vantaggi come ad esempio:



- riduzione di difetti in funzionalità appena sviluppate e la riduzione di bug nel caso in cui si voglia modificare una funzionalità già esistente.

- riduzione del costo di ricerca di un errore dato che avviene nelle fasi iniziali dello sviluppo di ogni modulo.
- miglioramento della progettazione, permettendo una migliore ristrutturazione del codice.
- quando integrato nella compilazione del codice aumenta la qualità del prodotto.

5.3 Integration Testing



Successivamente al completamento dello Unit Test, i singoli moduli vanno integrati e fatti collaborare. L'integration test si occupa di testare tale collaborazione, cioè porzioni di sistema composte da singoli moduli, che implementano solitamente un caso d'uso.
Gli obbiettivi dell'integration test sono:

- verificare che le funzionalità offerte dalla collaborazione di più moduli siano corrette.
- verificare che le performance di più moduli integrati siano accettabili.
- sondare il funzionamento dei moduli interni anche in parti non necessariamente esposte all'esterno.

5.4 System Testing

È una tecnica di testing che va a testare tutte le funzionalità dalla prospettiva dell'utente, sfruttando le istanze del sistema già funzionanti. Importantissimo per aver la certezza che, reso disponibile il sistema agli utenti, tutto funzionerà correttamente. Si eseguono sia

test funzionali che non (come nel caso del Chaos Monkey di Netflix, che è uno stress-test volto a verificare la resilienza del sistema).

5.5 Deployment Hooks

Dopo aver passato tutte le principali fasi di testing il software e quasi certamente funzionante e può essere messo in produzione.

L'automatizzazione del deployment è parte integrante del processo di continuous delivery. Generalmente avviene fornendo anche ai team non tecnici un'interfaccia grafica per gestire il rilascio di nuove funzionalità. Per ogni funzionalità che ha passato tutti i controlli viene creato automaticamente un "hook" (letteralmente gancio) da utilizzare per far scattare il suo deployment automatico.

5.6 Conclusioni

In questa sezione sono state indicate le fasi del continuous integration. Per ogni fase è necessario fornire un tool dedicato che automatizzi e strutturi la definizione di test, la loro verifica e il passaggio alla fase successiva.

Nel prossimo capitolo ci si concentrerà sul primo artefatto facente parte di un possibile framework di testing per microservizi, focalizzandosi sul design di un tool per lo Unit Testing, il suo sviluppo e l'automazione fornita all'utente.

Capitolo 6

Design di un tool di Unit Testing per Microservizi

Nel contesto dei microservizi, lo Unit Testing deve isolare un singolo microservizio e controllare che ogni operazione disponibile funzioni come ci si aspetti.

Lo Unit Testing generalmente è supportato da un framework per scrivere una test suite (un insieme di test). Definite le funzionalità da testare, avviene la vera e propria scrittura del codice necessario all'invocazione di una data funzionalità, insieme alle condizioni per definire il suo risultato positivo o negativo, i.e., se l'attuale implementazione della funzionalità è conforme a quanto ci si aspetta oppure no.

6.1 Testare un microservizio

In un sistema distribuito a microservizi, si potrebbe avere una commistione di vari linguaggi di programmazione che collaborano per fornire una funzionalità.

Serve quindi formalizzare un metodo per testare singolarmente porzioni di codice in linguaggi diversi. Tale metodologia di interazione è rappresentata dalle Remote Procedure Calls.

6.1.1 Remote Procedure Call

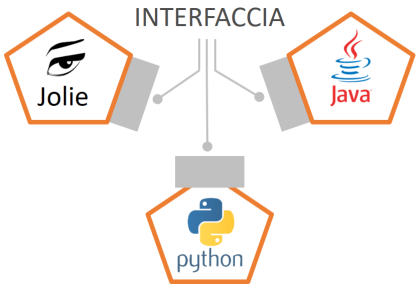
La metodologia più utilizzata per invocare una funzionalità in un sistema distribuito è tramite RPC, cioè Remote Procedure Call (letteralmente "chiamata di procedura remota").

RPC permette a un programma di richiedere una funzionalità (procedure) a un altro programma che sta eseguendo su una rete comune.

Il metodo più comune per offrire funzionalità tramite RPC è l'esposizione di un API, cioè un **interfaccia**, la quale definisce come invocare una certa procedura e come sono strutturati i dati trasmessi. Al lato pratico, è inoltre necessario definire con quale tecnologia avviene una certa RPC, e.g., tramite TCP/IP per il trasporto dei dati e HTTP per la loro formattazione. L'astrazione di interfacce e l'utilizzo di protocolli di comunicazione comuni permette l'indipendenza da una particolare tecnologia implementativa, rendendo possibile a componenti (e.g., microservizi) scritti con linguaggi diversi di interagire tra loro.

6.1.2 Come descrivere un microservizio

Per automatizzare l'interazione tra servizi, è fondamentale che la definizione delle interfacce e delle tecnologie di comunicazione siano processabili automaticamente. Per questo, nel design del tool di Unit Testing proposto, è importante avvalersi di tecnologie per:



- elencare le procedure invocabili
- dichiarare la struttura dei dati in ingresso e in uscita di ogni procedura.

- definire i protocolli di comunicazione utilizzati per invocare le procedure

Esempi comuni di tali tecnologie sono WSDL (Web Services Description Language) e REST (REpresentational State Transfer).

WSDL (Web Services Description Language)

Nato per i servizi web, WSDL definisce in un file XML:

- le funzionalità rese disponibili da un web service.
- come utilizzare queste funzionalità, stabilendo: il protocollo attraverso cui comunicare, il formato dei messaggi in entrata e il formato dei messaggi in uscita.
- come contattare il servizio, cioè l'endpoint.

Esempio di WSDL Un documento WSDL è composto da varie sezioni che definiscono:

I tipi utilizzati dalle funzionalità (nell'esempio tipi di dato già definiti in XMLSchema)

```
<message name = "SayHelloRequest">
   <part name = "firstName" type = "xsd:string"/>
</message>
<message name = "SayHelloResponse">
   <part name = "greeting" type = "xsd:string"/>
</message>
```

L'associazione dei tipi di dato dichiarati alle funzionalità fornite (in WSDL chiamate operation)

```
<portType name = "Hello_PortType">
   <operation name = "sayHello">
      <input message = "tns:SayHelloRequest"/>
      <output message = "tns:SayHelloResponse"/>
```

```
        </operation>
    </portType>
```

e infine le tecnologie di comunicazione utilizzate per contattare i servizi

```
    <binding name = "Hello_Binding" type =
        "tns:Hello_PortType">
        <soap:binding style = "rpc"
            transport =
                "http://schemas.xmlsoap.org/soap/http"/>
        <operation name = "sayHello">
            <soap:operation soapAction = "sayHello"/>
            <input>
                <soap:body
                    encodingStyle =
                        "http://schemas.xmlsoap.org/soap/encoding/"
                    namespace = "urn:examples:helloservice"
                    use = "encoded"/>
            </input>
            <output>
                <soap:body
                    encodingStyle =
                        "http://schemas.xmlsoap.org/soap/encoding/"
                    namespace = "urn:examples:helloservice"
                    use = "encoded"/>
            </output>
        </operation>
    </binding>

    <service name = "Hello_Service">
        <documentation>WSDL File for
            HelloService</documentation>
```

```
        <port binding = "tns:Hello_Binding"
        name = "Hello_Port" >
            <soap:address
                location = "http://www.examples.com/SayHello/"
            />
        </port>
    </service>
</definitions>
```

Riassumendo, nell'esempio di WSDL presentato:

- viene associata a una location l'interfaccia dell'operazione con struttura del dato in input e struttura del dato in output.
- vengono definiti i protocolli con cui comunicare (in questo caso SOAP HTTP)

REST (REpresentational State Transfer)

REST ha lo stesso obbiettivo di WSDL, ma si concentra sul gestire una comunicazione usando i verbi di HTTP [1].
Con REST si fornisce un servizio, mappando un indirizzo HTTP ad una determinata funzionalità.

Esempio REST Di seguito si riporta un esempio di descrizione REST, fatta tramite il tool Jersey[2]. Notare che REST, essendo un insieme di linee guida, non definisce un formato di specifica delle RPC, che è lasciato alla pratica degli implementatori.
Nell'esempio riportato nella pagina successiva viene descritto un servizio REST (servlet nel gergo di Jersey) insieme alla mappatura delle funzionalità fornite.

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
[2] https://jersey.github.io/

```
<servlet>
  <servlet-name>Jersey REST Service</servlet-name>
  <servlet-class>
  com.sun.jersey.spi.container.servlet.ServletContainer
  </servlet-class>
  <init-param>
    <param-name>
      com.sun.jersey.config.property.packages</param-name>
    <param-value>sample.hello.resources</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Jersey REST Service</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Nel File XML di definizione vengono specificate:

- L'esposizione di una servlet di nome "Jersey REST Service" dove viene indicato dove si potranno trovare le funzionalità (chiamate risorse) esposte dal servizio.
- La mappatura delle risorse, che in questo caso consiste nell'insieme di tutti gli indirizzi con prefisso /rest/.

Infine, ogni risorsa definisce un percorso con cui poter essere utilizzata. Se, ad esempio, vi sarà una risorsa che definisce il percorso "/hello", sarà raggiungibile all'URI /rest/hello.

```
@Path("/hello")
public class HelloResource {
    @GET
    @Produces(MediaType.TEXT_PLAIN)
```

```
    public String sayHello() {
        return "Hello Jersey";
    }
}
```

A sua volta la risorsa mapperà un metodo del programma (Java nel contesto Jersey) legato all'interfaccia. Ad esempio, il metodo sayHello() accetta HTTP GET e produce un dato di tipo "text-plain".

6.1.3 Requisiti

Ricapitolando, per comunicare con un microservizio e avere accesso alle sue funzionalità è necessario formalizzare quali di queste sono invocabili dall'esterno, (possibilmente) che tipi di dati vengono richiesti e restituiti e con che tecnologie invocarle.
Per questo, il tool di testing in oggetto deve considerare la possibilità di integrare una tecnologia tale da definire le interfacce dei microservizi nel modo più astratto (da un particolare tecnologia) e aperto possibile.

6.2 Come definire i test-cases

Altro punto chiave dell'esecuzione di test è la progettazione di un linguaggio per definire tutti i test-case.
Per test-case si intente un singolo test composto dalla dichiarazione di un valore in input, l'identificazione della funzionalità da testare, e la prova del risultato.

Considerando il contesto dei microservizi, un linguaggio per definire test-case per test unitario dovrebbe offrire come minimo:

- un metodo per definire l'input con cui si vuole testare una funzionalità.
- un metodo definire l'invocazione di questa funzionalità, trovando un modo per isolarla da qualsiasi dipendenza.

- un metodo per specificare la procedura di acquisizione del risultato dell'invocazione e la sua valutazione di positività.

6.3 Esecuzione dei Test

Dopo la specifica e in previsione dell'implementazione del tool di Unit Testing in oggetto, è necessario considerare la vera e propria esecuzione dei test

6.3.1 Interazione con un Sistema Distribuito

Per sommi capi ogni sistema distribuito si divide in due parti: un client che invoca una funzionalità e un server che fornisce tale funzionalità. Per eseguire un test su ogni funzionalità di un microservizio, è necessario avere un client che riesca a interfacciarsi in modo corretto con ogni funzionalità testata.

Generazione Client Vi è quindi la necessità di generare una batteria di client configurati per comunicare in modo corretto. Nel design proposto, tali client possono essere generati dinamicamente sulla base dell'interfaccia che descrive un microservizio.

6.3.2 Esecuzione dei Test

All'atto pratico, per eseguire i test è necessario un orchestratore che riesca a eseguire ogni test-case di una data suite, oltre a gestire l'esito di ogni test, producendo un output riassuntivo di facile consultazione da parte dell'utente.

6.4 Conclusioni

In questo capitolo è stata analizzata la comunicazione attraverso un sistema distribuito e quindi i requisiti necessari per identificare le varie funzioni da testare.
Sono stati analizzati i requisiti che deve avere un linguaggio per scrivere test case.
Successivamente ci si è concentrati su come dovrebbe avvenire il test vero e proprio, sfruttando client generati ad hoc e un orchestratore generale dei test.

Nei capitoli successivi si entrerà nel dettaglio di ognuno dei punti precedenti, spiegando le scelte implementative riguardo il linguaggio utilizzato per descrivere interfacce e test, dettagliando infine la generazione dei client e l'implementazione dell'orchestratore dei test.

Parte III

Sviluppo

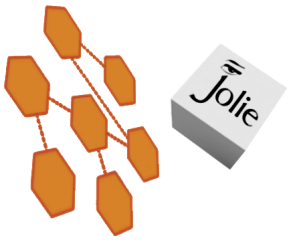
Capitolo 7

Jolie

In questo capitolo verrà spiegato perché si è deciso di adottare Jolie per scrivere i test case e anche per sviluppare l'orchestratore dei test.

7.1 Perché Jolie

7.1.1 Cos'è Jolie



Jolie è il primo linguaggio di programmazione service-oriented che fornisce supporto nativo per microservizi. Per nativo si intende che il blocco base su cui si struttura un'applicazione Jolie, non è un oggetto o una funzione, ma un servizio.
Il codice di Jolie è sempre contenuto in un servizio, che può eseguire in locale, ma all'occorrenza spostarsi su un server in remoto.
Un'applicazione scritta in Jolie è un sistema distribuito fatto da vari microservizi che collaborano per portare a termine un task.

Linguaggio Protocol-Agnostic Jolie è indipendente da protocolli di comunicazione e due microservizi per comunicare possono utilizzare vari protocolli come SODEP, SOAP o HTTP. Aprire una connessione tra due microservizi scritti in Jolie è questione di poche righe di codice.

Con Jolie si può facilmente costruire un'applicazione fatta da mircoservizi che comunicano tra loro con protocolli diversi.

L'aspetto fondamentale che ha decretato la scelta di Jolie, sia per la descrizione delle interfacce che per la definizione dei test, è che il linguaggio fornisce nativamente i costrutti necessari alla specifica di entrambe.

7.1.2 Interfacce e Porte native

Com'è stato spiegato nei capitoli precedenti, il framework proposto necessita di un linguaggio in grado di descrivere l'interfaccia di un microservizio e il canale di comunicazione (d'ora in poi chiamato porta).
Jolie supporta nativamente i concetti di Interfaccia e Porta rendendo facile l'interazione con microservizi dei linguaggi più disparati..
La facilità di integrazione consentita dal linguaggio rappresenta un ulteriore motivo che ha fatto ricadere la scelta per definire i test ed eseguirli su Jolie.

7.2 Struttura del codice

Si entrerà ora nel dettaglio di come è strutturato il codice in Jolie, in particolare di come è strutturata un'interfaccia e una porta.

7.2.1 Hello, World!

Come da documentazione[1] iniziamo dal classico esempio della stampa a terminale di "Hello, World"

```
include "console.iol"

main {
   println@Console( "Hello, World!" )( )
```

[1] https://docs.jolie-lang.org/documentation/getting_started/hello_world.html

```
}
```

Analizzando questo primo esempio si può notare:

- `include "console.iol"` dove "console.iol" è un interfaccia per utilizzare le funzionalità del terminale (come normali chiamate a servizi).
- `println@Console( "Hello, World!" )( )` che invoca col messaggio (la stringa) "Hello, World!" la funzionalità println del servizio Console. In Jolie, le funzionalità offerte dai servizi vengono chiamate operation (operazioni). Di seguito si userà tale termine per indicare le funzionalità di un microservizio.

7.2.2 Behaviour & Deployment

In Jolie un servizio è descritto da due parti separate, chiamate Behaviour e Deployment.

Behaviour Questa parte si occupa di definire l'implementazione delle funzionalità che offre un microservizio, primitive comportamentali come costrutti di comunicazione e di calcolo.
Essendo la parte comportamentale, non si occupa di definire come avverrà la comunicazione con eventuali microservizi da cui dipenderanno le varie operazioni.

Ad esempio un microservizio Client che "chiede al servizio Calculator di raddoppiare il valore 3 e restituire il risultato" ha la seguente forma

```
main{
    raddoppia@OutCalculator( 3 ) ( risultato );
    println@Console( risultato )( )
}
```

Si può notare la struttura del behaviour che tramite la Porta `OutCalculator` (definita nel deployment) comunica con il servizio Calculator e richiede l'operation `raddoppia` passando come input 3.
Il servizio Calculator raddoppierà il valore e restituirà un risultato che verrà salvato nella variabile `risultato`. Poi verrà stampato a console questo risultato.

Calculator (Server) Il lato server della comunicazione sarà il servizio Calculator e implementerà il comportamento dell'operazione raddoppia.

```
main{
   raddoppia( request )( response ){
      response = request * 2
   }
}
```

La parte behaviour del server definisce un'operazione `raddoppia` che memorizza il valore del messaggio ricevuto dal client in una variabile `request`. Questa variabile viene moltiplicata per 2 e memorizzata nella variabile `response`, la quale verrà mandata come risposta all'invocatore di `raddoppia`.

Il comportamento del client è quindi complementare a quello del server. Come si può notare il client invoca l'operazione `raddoppia`, esposta dal server, sulla porta `Calculator` e si mette in attesa di risposta. Il server (Calculator) riceverà il valore 3 mandato dal client, lo raddoppierà e manderà indietro il valore 6. Il client riceverà la risposta e continuerà l'esecuzione stampando questo la risposta a console.

Deployment La comunicazione tra Client e Calculator avviene attraverso delle Porte di comunicazione. In Jolie si distinguono due tipi di porte:

- input port: espone le operazioni implementate verso altri servizi che possono invocarle.
- output port: definisce come invocare le operazioni di altri servizi.

In generale le porte definiscono 3 elementi fondamentali:

- location: ad esempio un indirizzo TCP/IP, che identifica indirizzo e porta presso cui trovare un servizio o presso cui un servizio espone le proprie operation.
- protocol: definisce la codifica dei dati mandati o ricevuti tramite una porta (ad esempio HTTP).

- interface: una collezione di operazioni One-Way (operazione che richiede solo un input) e Request-Response (operazione che richiede un input e un output). In questa collezione di operazioni vengono definite le strutture dei dati in ingresso e in uscita per ogni operazione.

Interfaccia Nell'esempio precedente, per Client e Calculator, l'interfaccia dichiarava un'operazione Request-Response raddoppia che accettava in ingresso un valore intero e restituiva un valore intero.

```
interface CalculatorInterface {
    RequestResponse:
      raddoppia( int )( int )
}
```

Per utilizzare questa interfaccia nella parte Deployment del servizio, sia Client che Calculator dovranno includerla.

Calculator (Server) Lato server per esporre l'operation `raddoppia` dovremmo includere il file CalculatorInterface.iol contenente l'interfaccia e definire una input port.

```
include "interface.iol"

inputPort InCalculator {
    Location: "socket://localhost:8000"
    Protocol: sodep
    Interfaces: CalculatorInterface
}
```

Questa inputPort esporrà l'interfaccia CalculatorInterface (quindi la singola RequestResponse raddoppia) all'indirizzo IP localhost sulla porta 8000.
La comunicazione avverrà tramite il protocollo SODEP (Simple Operation Data Excange Protocol), un protocollo binario sviluppato apposta per Jolie per fornire un protocollo semplice, sicuro ed efficiente per comunicazioni su servizi.

Client Lato client per utilizzare le operazioni esposte dal Calculator (server) sarà necessario includere la stessa interfaccia del server e una outputPort per invocare le operation.

```
incude "CalculatorInterface.iol"

outputPort OutCalculator {
    location: "socket://localhost:8000"
    Protocol: sodep
    Interfaces: CalculatorInterface
}
```

Questa outputPort permette di comunicare con il servizio in ascolto sull'indirizzo IP locale sulla porta 8000, che implementa le operazioni definite nell'interfaccia `CalculatorInterface`.

7.2.3 Interfacce e tipi di dato nel dettaglio

Interfacce Si entra ora nel dettaglio di come è strutturata un'interfaccia, fondamentale per capire perchè il tool sviluppato potrà essere usato per testare qualsiasi microservizio che utilizzi protocolli offerti da Jolie.
Un'interfaccia può descrivere 2 tipi di operazioni Jolie:

One-Way: riceve un messaggio.
Si scrive nella forma `ow_name( rq )`

Request-Response: risponde o riceve un messaggio e invia una risposta.
Si scrive nella forma `rr_name( rq )( rs )`

Si può fare un esempio un po più complesso di interfaccia.

```
interface CalculatorInterface {
    RequestResponse:
        moltiplica( PerRequest )( int )
}
```

In questa interfaccia si definisce una operazione che moltiplica due numeri. C'è però una cosa in più rispetto all'interfaccia vista prima: come tipo della request abbiamo utilizzato `PerRequest`: un tipo di dato, di seguito chiamato data type.

Data Type Con Jolie posso definire 3 tipi di data type

Basic data type: tipo di dato base.

Si scrive nella forma

```
type Name: string
```

Subtree data types: un tipo può avere una lista di sotto-nodi data type.

Si scrive nella forma

```
type Name: basic_type {
.sottonodo1: basic_type
...
.sottonodoN: basic_type
}
```

Specificando **cardinalità**: si può specificare qual'è il numero massimo di elementi per un certo tipo.

Si scrive nella forma

```
type Name: basic_type {
.sottonodo1[0, 1]: basic_type
}
```

Tornando al tipo dell'esempio `PerRequest` nell'interfaccia `CalculatorInterface`

```
type PerRequest: void {
    .number[2,*]: int
}

interface CalculatorInterface {
    RequestReponse: moltiplica( PerRequest ) ( int )
}
```

PerRequest è un data type che definisce un sotto-nodo `.number[2,*]: int` perché specifica che vengono accettati valori con un sotto-nodo number contenente almeno 2 numeri interi, per poterli poi moltiplicare

7.3 Conclusioni

In questo capitolo abbiamo illustrato le principali funzionalità offerte dal linguaggio Jolie per sviluppare applicazioni orientate ai servizi.
In questo linguaggio vengono implementate come funzionalità native:

- Interfacce, per descrivere le varie operation di un servizio e la struttura dei dati passati in input e in output.
- Porte, per definire con che protocollo e verso che location indirizzare una richiesta. Inoltre nella porta viene definita anche un'interfaccia per sapere quali operazioni sono disponibili.

Nel prossimo capitolo verrà descritto come è stato sviluppato l'orchestratore dei test e come andranno scritti i test case per ogni operazione.

Capitolo 8

Sviluppo Unit Test (JTS)

8.1 Schema JolieTestSuite e requisiti

8.1.1 Requisiti

Come visto nei capitoli precedenti, Jolie offre due costrutti molto importanti per fare testing di microservizi: Interfacce e Porte.

Supportando nativamente la comunicazione con molte tecnologie di comunicazione, per comunicare dal framework di testing scritto in Jolie a un qualsiasi microservizio è solo necessario avere la definizione di due componenti di tale microservizio:

- un'**interfaccia** ed eventualmente di **data type** appositi per descrivere le operazioni
- una **porta** per comunicare con questo microservizio

Quindi dovrà essere necessario solamente scrivere poche righe di codice (Jolie) per rendere testabile qualsiasi microservizio.

8.1.2 Schema JolieTestSuite

Le fasi del test unitario di un microservizio saranno divise in due parti principali:

- generazione dei client

- isolamento del microservizio dalle dipendenze
- esecuzione dei test case

Il codice sorgente di tutta la JolieTestSuite per Unit Test si può trovare al link https://github.com/sanfo3855/JolieTestSuite[1]

8.2 Generazione dei Client

8.2.1 Obbiettivo: Surface e Client

Si è voluto strutturare il client come un intermediario tra la l'orchestratore dei test e il micorservizio. Il client autogenerato implementerà un'operation (run) la quale:

1. riceverà il messaggio di input come request
2. effettuerà la chiamata all'operation sul servizio da testare (e passerà l'input)
3. riceverà l'output dall'operation sotto test
4. ritornerà l'output all'orchestrator dei test.

Il tutto inserito all'interno di un'operation `run( request )( response )`.

```
main {

  run( request )( response ){
    grq.request_message = JDEP_a;
    grq.name = "/mainIn/incNum";

    goal@GoalManager( grq )( res5 );

    expectedResult = 10;
```

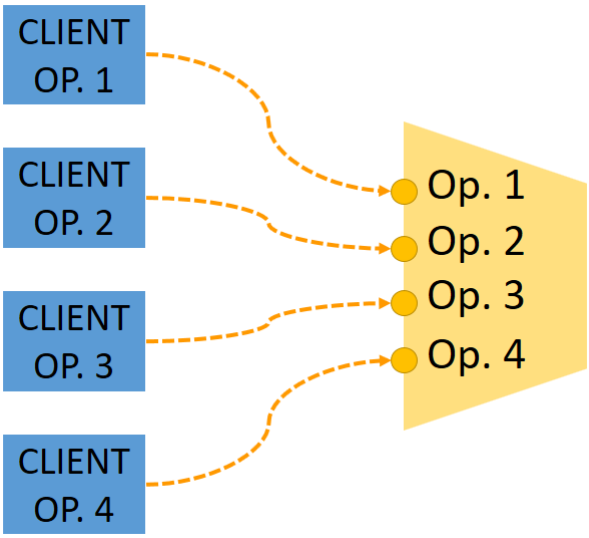
[1]https://github.com/sanfo3855/JolieTestSuite

```
    if( res5 != expectedResult ){
      fault.message = "res: "+res+" expRes: "+expectedResult;
      fault.faultname = "Fault incNum";
      throw( ExecutionFault, fault)
    }

  }
}
```

8.2.2 Generazione

Linguaggio Service-Oriented Per generare un client per ogni operazione sarà necessario fornire un'interfaccia che definisce la lista delle operazioni e gli eventuali data type e la porta con cui il microservizio offre queste operazioni.



Ottenere le informazioni necessarie Creato un microservizio con queste tre informazioni, sarà possibile sfruttare l'operation `getInputPortMetaData` dell'API di Jolie `MetaJolie` per ottenere una struttura dati che descrive una certa input port (location, protocol, interfacce con relative operation e type).
Questa struttura dati per ogni inputPort descriverà tutte le sue interfacce e le operation che espone.

Generare la Surface La surface non sarà altro che un file chiamato **testport_surface.iol** dove sarà definita interfaccia e porta per comunicare con il microservizio da testare. Ogni client

dovrà fare l'include di questa surface.

La surface può essere generata dando in pasto a `getSurface` dell'API `Parser` la struttura dati appena ricevuta da `getInputPortMetaData`.
Un esempio di **testport_surface.iol** auto-generata può essere:

```
interface mainInInterface {
RequestResponse:
  incNum( int )( int ),
  twiceNum( int )( int ),
}

outputPort mainIn {
  Protocol:sodep
  Location:"socket://localhost:13000"
  Interfaces:mainInInterface
}
```

Generazione Clients Sempre dalla stessa struttura dati da cui abbiamo generato la surface, si possono estrarre, per ogni porta, le operation disponibili. Per ogni inputPort del microservizio si andrà a creare una cartella contenente i client auto-generati.
Come spiegato, ognuno dei client generati implementerà un'operation per testare ogni singola operazione del servizio sotto test.
L'operazione con cui "attivare" un dato client riceverà come request un eventuale messaggio di input, e manderà come response un eventuale messaggio di output.
La struttura semplificata di un client auto-generato è la seguente:

```
include "testport_surface.iol"

...

main{
```

```
    run( request )( response ) {
      scope( test ) {
        ...
        incNum@mainIn( request )( response )
      }
    }
}
```

Come prima cosa vi è l'`include` della **testport_surface.iol**, in modo tale da poter usare l'outputPort mainIn per comunicare con il servizio.
Successivamente per ogni client viene generata un'operation `run(request)(response)`. All'interno di questa run viene invocata l'operation `incNum` sulla porta `mainIn` disponibile tramite la surface inclusa in cima.
L'orchestrator dei test manderà un messaggio di input al `run`, questo messaggio verrà usato da `incNum` per contattare il servizio, il quale darà una risposta che verrà memorizzata su response. Questa response, verrà poi mandata come risposta all'orchestrator dei test.

8.3 Scrittura test case ed Esecuzione

8.3.1 Obbiettivo

I client auto-generati riceveranno un messaggio di input, contatteranno il servizio e manderanno all'orchestratore dei test una risposta.
Il servizio GoalManager fornirà l'implementazione che eseguirà i "Goal" scritti nei test case.

8.3.2 Scrittura di un Test Case

Prima di spiegare come funziona il GoalManager è bene spiegare come è strutturato un Goal:

1. Richiesta per il goalManager (messaggio di input + target operation)

```
    grq.request_message = 3;
    grq.name = "/mainIn/incNum";
```

2. chiamata al GoalManager (invio request, ricezione di una response)

```
goal@GoalManager( grq )( response );
```

3. controlli da fare sulla response e throw di un fault se i controlli non hanno risultato positivo

```
expectedResult = 10;
if( res5 != expectedResult ){
  fault
  throw( ExecutionFault, fault)
}
```

Il tutto verrà poi inserito all'interno di un'operation `run( request )( response )` che astrarrà l'esecuzione dal contenuto di ogni test case.
Un test case non sarà altro che un file di testo scritto in Jolie che sfrutta i servizi offerti dal GoalManager.

Gruppi di test case Ogni file potrà contenere uno o più test case (request, chiamata al goal manager, controllo response).
La parte di controllo della response potrebbe essere più complessa di un confronto con un valore atteso. Si potrebbe dover processare una struttura dati per vedere se contiene gli output necessari ma anche confrontare più response di più test case tra loro.

Generalmente la linea guida da seguire dovrebbe essere raggruppare gruppi test case in file separati e fare un file `init.ol` per lanciare ogni gruppo di test case.

```
## init.ol ##
main {
  run( request )( response ){
```

Questa chiamata al GoalManager si occupa di eseguire il file `incnum-test.ol` con i test relativi all'operation `incNum`

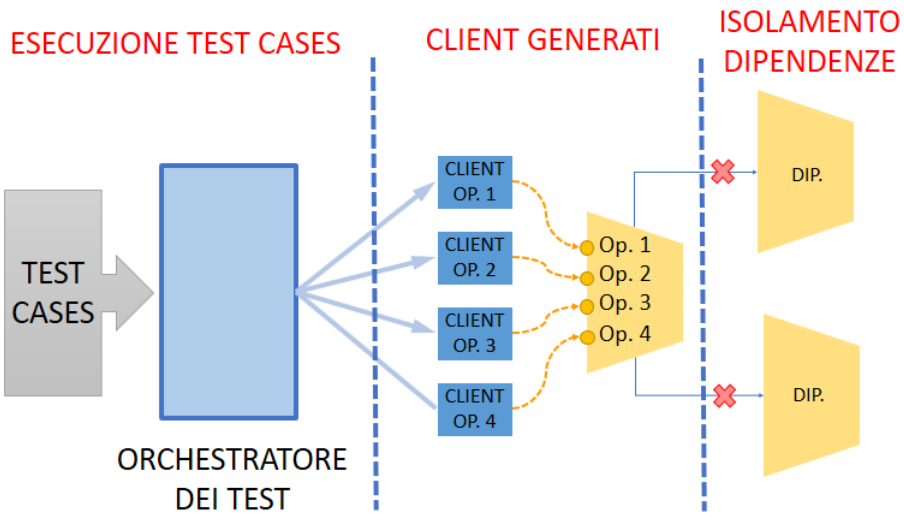
```
    grq.name = "incnum-test";
    goal@GoalManager( grq )( res );
```

Questa invece esegue i test relativi all'operation `twiceNum`

```
    grq.name = "twicenum-test";
    goal@GoalManager( grq )( res )
  }
}
```

I test case saranno organizzati sotto forma di albero di test e sotto-test.
Come test case di partenza ci dovrà essere per forza il file l'init da cui si snoderanno una serie di sotto-test; ogni test potrà avere a sua volta dei sotto-test.

8.3.3 Esecuzione: Orchestrator dei test e GoalManager



L'orchestrator dei test (main_test_suite.ol) è stato sviluppato come servizio principale di ogni esecuzione.
Con l'orchestrator verra inizializzata l'istanza di testing e quindi verranno fatte chiamate al

GoalManager per ogni test case o sotto-test scritto nell'init.
Si vedrà poi che il GoalManager è stato sviluppato in modo che possa riceve come richiesta sia la chiamata a un sotto-test, sia la chiamata a un client specifico di un servizio da testare.

GoalManager Il GoalManager espone un'unica operation `goal`, che riceve il nome di un file contenente una batteria di test case (siano essi chiamate a sotto-test o test di un'operazione). Ogni chiamata `goal@GoalManager` è divisa in:

1. Creazione di un file temporaneo composto da un template di inizializzazione, unito al codice scritto all'interno di ogni gruppo di test case.
   Nel caso in cui la chiamata al GoalManager abbia come request una operation specifica, al template di inizializzazione verrà aggiunto il client auto-generato (come spiegato nella sezione precedente di Generazione dei Client).

2. Questo file (che è un vero e proprio servizio) temporaneo verrà incorporato in una porta chiamata `Goal` che esporrà sempre solo una operation `run(request)(response)`. Dopo aver caricato a runtime il servizio temporaneo, verrà chiamata l'operation `run@Goal(request)(response)`.

Dato che nella run di un test (o di un sotto-test) potrà essere invocato tramite il GoalManager un eventuale sotto-test, si scenderà nella struttura ad albero dei sotto-test finché non verrà eseguito un client (test vero e proprio su di una operazione), per poi risalire alla diramazione più vicina. L'esecuzione terminerà quando verrà terminata l'esecuzione di tutti i sotto-test dell'init, cioè il test radice.

In caso di fallimento di un test su un client, verrà lanciato un errore che ricorsivamente manderà in errore tutti i test al di sopra fino a far fallire l'init.

Nella prossima pagina si può vedere un esempio di output del tool

```
---> RUNNING TEST
 TESTING init...
    TESTING incnum-test...
       TESTING /mainIn/incNum...
       SUCCESS: /mainIn/incNum
    SUCCESS: incnum-test
    TESTING twicenum-test...
       TESTING /mainIn/twiceNum...
       SUCCESS: /mainIn/twiceNum
    SUCCESS: twicenum-test
 SUCCESS: init
```

Appendice sull'esecuzione Fin'ora si può notare che l'esecuzione dei test non è molto automatizzata. Di seguito si elencano i passaggi obbligatori necessari all'esecuzione di un test-case.

8.4 Emulazione delle dipendenze

8.4.1 Obbiettivo: dipendenze

Per fare un vero e proprio unit test, il microservizio sotto test deve essere isolato da eventuali dipendenze. Un microservizio avrà una dipendenza quando andrà a invocare un operation esterna esposta da un altro microservizio, quindi dipenderà da esso.
Per fare unit testing il microservizio deve essere completamente isolato, sia dall'ambiente che dalle sue dipendenze.
Questo problema viene introdotto ora per chiarezza, dato che il processo di generazione dei client ed esecuzione resta invariato. La fase di scrittura dei test varierà (anche se di poco).

8.4.2 Generazione surface

L'emulazione delle dipendenze passerà per una fase di generazione di file particolari di formato "depservice"; il servizio `getDependenciesPort.ol` si occuperà di generarli per ogni dipendenza.

Generazione Ogni microservizio con dipendenze ha delle `outputPort` verso qualche altro servizio.

I file depservice verranno generati a partire dalle outputPort del microservizio da testare. Ognuna di queste outputPort necessiterà di un finto servizio da chiamare con una finta implementazione dell'operazione invocata.

Con l'operation `getMetaData` implementata dall'API di Jolie "MetaJolie" si può ottenere una struttura dati che descrive tutti i metadata di un servizio (InputPort, OutputPort, Interfacce, Data Type).

Andremo a generare un file "depservice" per ogni outputPort in modo che a runtime non sia necessario aver in esecuzione un'istanza di ogni dipendenza e che il microservizio sotto test non vada in crash.

```
## dep2Out.depservice ##
interface dep2Interface {
RequestResponse:
  twice( int )( int )
}

inputPort dep2Out {
  Location: "socket://localhost:13002"
  Protocol: sodep
  Interfaces: dep2Interface
}
```

Ad esempio in questo caso viene generato un depservice per emulare l'operation `twice` richiesta alla outputPort `dep2out`.

Si può notare come l'outputPort sia diventata una inputPort. La motivazione è che, a runtime, il goal che necessiterà di questa dipendenza avrà l'`include` del relativo depservice e il goal stesso fornirà al microservizio una porta in input da invocare (senza la necessità che il servizio "dep2" sia attivo).

Scrittura del test case in caso di dipendenza Come spiegato prima un semplice test case andrebbe a:

(I) definire una richiesta,

(II) invocare `goal@GoalManager` passando tale richiesta,

(III) eseguire controlli sulla risposta.

Dato che nel momento in cui si andrà a invocare il GoalManager verrà eseguito il client relativo a una data operazione con dipendenza, se la dipendenza non fosse attiva verrebbe sollevato un errore.
Per mantenere il microservizio sotto test isolato, in questo test case servirà aggiungere due cose:

- `include "<nomeOutputPort>.depservice"`
- implementazione banale in parallelo al GoalManager dell'operation che servirebbe da parte della dipendenza.

Il test case diventerà quindi

```
include "dep2Out.depservice"
main {
  run( request )( response ){
```

Definizione della richiesta uguale all'esempio nella sezione 8.3.2 (Scrittura di un Test Case)

```
    { goal@GoalManager( grq )( res5 ) | twice( request )(
       response ){ response = 6} };
```

Controlli uguali all'esempio nella sezione 8.3.2 (Scrittura di un Test Case)

```
  }
}
```

Si noti l'`include` in cima del file che implementa la porta `dep2out` e l'implementazione di simulazione che ritorna un risultato diretto dell'operazione necessaria `twice`.

8.5 Conclusioni

In questo capitolo è stato spiegato come è stata sviluppata la generazione dei client, come è deve essere fatta la dichiarazione dei test case e come è stata sviluppata l'esecuzione, con anche l'emulazione delle dipendenze

Nel prossimo capitolo si vedrà come è stato automatizzato questo processo, che ora deve essere fatto manualmente passando per gli step di: (I) generazione dei client, (II) generazione delle dipendenze, (III) esecuzione del microservizio da testare, (IV) esecuzione dell'orchestatore dei test. Tralasciando la parte di scrittura dei test case che resta manuale per dare più libertà nell'effettuare controlli complessi sui risultati dei test.

Capitolo 9

Sviluppo JoUnit

9.1 Introduzione al tool

Il tool JoUnit, disponibile al link https://github.com/jolie/jounit[1], si occupa di automatizzare e isolare completamente dall'ambiente di sviluppo tutti i passaggi per eseguire i test. L'automatizzazione e l'isolamento avviene sfruttando due strumenti:

- Git
- Jocker (Docker)

9.1.1 Cos'è Git

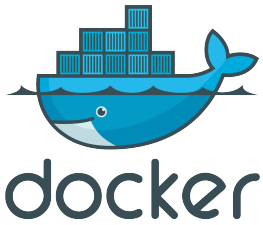
Git è un sistema di controllo versione distribuito, che permette a team di più persone di lavorare sullo stesso progetto sfruttando degli spazi dedicati, chiamati repository. In ogni repository è presente un ramo di sviluppo (branch) master (principale) col software aggiornato e funzionate all'ultima versione, in aggiunta al master, si possono trovare branch di sviluppo. Git si inserisce bene nei processi di sviluppo distribuiti che devono essere gestiti da un tool di continuous integration.
Più nello specifico, parlando di unit test, con git potrebbe essere automatizzato l'esecuzione

[1]https://github.com/jolie/jounit

dei test case.
Ad esempio potrebbe esser eseguita la catena di operazioni per lo unit test all'aggiornamento di un repository oppure prima del merge con il branch principale.

9.1.2 Cos'è Jocker (e Docker)

Sempre prendendo spunto da ciò che viene detto in "The DevOps 2.0 Toolkit", tutte le operazioni che dovranno eseguire i tool di testing e deployment in un sistema distribuito, saranno più semplici se supportate da un sistema di containerizzazione come Docker[2].



Docker è un sistema che punta a facilitare e automatizzare il deployment (cioè consegna al cliente, con installazione e messa in funzione) di un applicazione o di un sistema.
Docker ha scardinato il deployment fatto tramite macchine virtuali, tool per il management di sistema complessi e la necessità spesso di diverse librerie per il funzionamento di una applicazione.
La difficoltà era data dal dover configurare le macchine virtuali con all'interno parti di un sistema distribuito, che richiedeva tempo e l'utilizzo di tool complessi e poi collegare in rete tutte le porzioni del sistema. Data la complessità era difficile anche far manutenzione del sistema e in casi di crash e problemi serviva intervento manuale per rimettere in funzione il tutto.

Innovazione di docker Docker va a eliminare tutta questa fase complessa di configurazione prima del deployment tramite il concetto di Container: il più piccolo set di funzioni per far funzionare un software.
Se per esempio serve far funzionare un servizio in Java, esiste un container per Java che altro non è che un sistema operativo Linux ridotto alle funzioni minime per eseguire codice Java.

Jocker e Jolie Anche per microservizi Jolie è possibile fare il deployment all'interno di un Container Docker. Inoltre, con Jolie è molto semplice orchestrare tutto un sistema Docker,

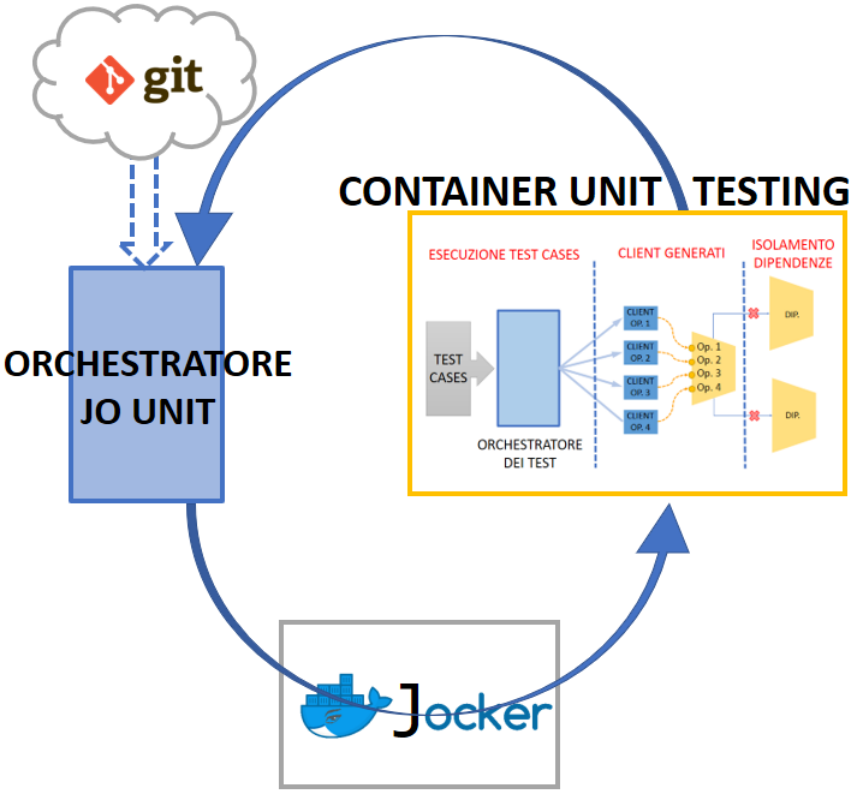
[2]https://www.docker.com/what-docker

dato che è stato sviluppato un wrapper completo delle API di Docker: Jocker. Jocker offre la possibilità di usare le API di Docker in modo semplice e immediato direttamente all'interno di un servizio scritto in Jolie.

9.2 Sviluppo JoUnit

9.2.1 Obbiettivo e Requisiti

Con JoUnit si è voluto automatizzare il deployment di unit test, riducendo la fase di unit testing a scrittura dei test case ed esecuzione. Idealmente, JoUnit sarà il primo di una serie di tool per automatizzare e rendere modulare un processo di continuous integration e deployment di un sistema distribuito a microservizi.



Requisiti Per utilizzare questo tool sarà necessario:

- avere il codice del microservizio con relativi test case in un repository compatibile con git.
- aver installato docker (e anche avere il container di Jocker attivo), con cui si potrà avere un ambiente stabile e isolato per fare unit testing di singoli microservizi

Struttura JoUnit Questo tool è stato sviluppato in modo da

1. ottenere automaticamente il codice necessario
2. creare un container isolato e pronto per lo unit testing
3. caricare al suo interno il microservizio e i test case
4. eseguire i test

Il tool può essere diviso in:

- orchestratore
- container per il testing

9.2.2 Orchestratore

Per gestire questa serie di operazioni di generazione e lancio del container, è stato sviluppato un servizio `orchestrator_jo_unit.ol`.
L'orchestrator è diviso in 2 parti: inizializzazione e implementazione dell'operation per stampare i risultati.

Inizializzazione in questa fase vengono definite le porte per comunicare con Jocker e creare il container per eseguire il test e ricevere i risultati dal container.
Il lancio del tool avviene eseguendo
`jolie orchestrator_jo_unit.ol indirizzo-al-repository`
L'indirizzo può essere sia una percorso locale (assoluto o relativo) sia un URL, in ogni caso è richiesto un indirizzo di un repository compatibile con git.

L'inizializzazione è composta di 3 fasi:

1. download repository in una cartella temporanea

```
git clone <indirizzo-al-repo> ./tmp
```

2. creazione del dockerfile, cioè di un file per configurare un container Docker. In questo dockerfile è possibile specificare il container base, aggiungere file programmaticamente, settare l'ambiente con parametri.

```
## dockerfile ##
FROM jolielang/unit-test-suite
WORKDIR /microservice
COPY ./tmp .
ENV ODEP_LOCATION="socket://localhost:8001"
```

`FROM` stabilisce l'immagine base per il container, quest'immagine è stata configurata appositamente per eseguire i test case (verrà spiegato nel paragrafo Container per il Testing). `WORKDIR` e `COPY` servono rispettivamente per posizionarsi nella directory `/microservice` e copiarci dentro il contenuto del repository scaricato in `tmp`
`ENV` definisce una variabile d'ambiente `ODEP_LOCATION` all'interno del container. Viene utilizzato questo metodo per parametrizzare la location a cui il container dovrà mandare i risultati e dove ci sarà in ascolto l'`orchestrator_jo_unit`.

3. compilazione del dockerfile, creazione del container, esecuzione del container

```
build@Jocker( rqImg )( response );
createContainer@Jocker( rqCnt )( response );
startContainer@Jocker( crq )( response );
```

Serie di operation esposte da Jocker per compilare un'immagine, creare un container, eseguire un container.

Stampa dei risultati e pulizia L'orchestrator esporrà una serie di operazioni wrapper delle API della console di Jolie tra cui print e println. Queste operation saranno utilizzate all'interno del container per stampare a terminale, sarà poi spiegato come verrà intercettata ogni operazione e mandata all'orchestrator. L'orchestrator si "fingerà" console e quindi esporrà tutte le operazioni del servizio "console.iol" di Jolie

```
main{
  [println ( request )( response ){
     println@Console( request )( )
  }]{
  ..Controlli..
  }

  [..Altre API di Console..]
}
```

In questa porzione di codice possiamo notare che c'è una parte di codice `..Controlli..`
Dato che tutte le stampe dei risultati avvengono all'interno del container con l'operation `println`, dopo ogni stampa è stato pensato di controllare l'output. Nel momento in cui verrà intercettato l'output `SUCCESS: init` il test sarà terminato correttamente e verranno eseguite le operazioni per fermare ed eliminare il container e l'immagine del test appena terminato.

```
stopContainer@Jocker( rmCnt )( response );
removeContainer@Jocker( rmCnt )( response );
removeImage@Jocker( rmImg )( response );
```

Anche nel caso in cui ci sia un output di fallimento del test `TEST FAILED! : init`, verrà comunque fermato il container ed eliminato insieme alla sua immagine.

9.2.3 Container

È stato progettato e sviluppato un container Docker per automatizzare l'esecuzione dei test. Questo container è stato costruito a partire dal container base per eseguire codice in Jolie.
Da quel container è stato compilato questo dockerfile

```
FROM jolielang/jolie-deployer
WORKDIR /
RUN git clone https://github.com/sanfo3855/JolieTestSuite
```

In questa prima porzione di dockerfile si può vedere:
`FROM` che indica su cosa viene basata questa immagine
`WORKDIR` e `RUN git clone..` per scaricare la versione più aggiornata del codice sorgente per eseguire gli unit test

```
COPY console.iol /JolieTestSuite
COPY console-imp.ol /JolieTestSuite
COPY launch.sh /
CMD ["/bin/bash","/launch.sh"]
```

`COPY` di `console.iol` e `console-imp.iol`, 2 file per reindirizzare gli output dell'orchestrator dei test (normalmente interni al container e quindi non visibili) all'esterno del container verso l'`orchestrator_jo_unit.ol` che, come abbiamo visto prima, re-implementano le operation di console.
`COPY` del file `launch.sh` che sarà poi utilizzato nel `CMD` successivo per stabilire che cosa verrà eseguito al lancio di questo container.

launch.sh è un semplice script che esegue in successione i comandi per generare i file necessari al test ed eseguire tutti i test cases.
Viene eseguito in successione:

1. il servizio per generare i client necessari al `GoalManager` per testare le singole operazioni
2. il servizio per generare i file ".depservice" necessari all'emulazione delle dipendenze
3. il servizio da testare che deve essere in esecuzione e si chiamerà sempre `main.ol` come requisito
4. il servizio che fa partire il test dall'init

Come è stato spiegato nella sezione precedente sull'Orchestratore del Container, verrà utilizzata questa immagine come base a cui verrà aggiunto il codice necessario all'esecuzione del

microservizio e i test case.
Poi l'orchestratore del Container crea e fa partire il container il quale automaticamente genera i file, esegue il micorservizio ed esegue i test case.

console.iol e consol-impl.iol Questi file andranno a intercettare le chiamate al servizio Console, che normalmente dovrebbe stampare nella console locale (interna al container).
Ogni invocazione di operation di console verrà reindirizzata all'esterno del container verso l'`orchestrator_jo_unit` il quale si occuperà di stamparli nel terminale del sistema operativo host di Docker. Sono posizionati nella stessa cartella del servizio che lancia i test, in modo da intercettare solo i suoi output.

console.iol In maniera predefinita, nel momento in cui vi è un `include "console.iol"`, l'interprete Jolie va prima a cercare questo file nella directory d'esecuzione e poi dentro la directory di installazione dell'interprete stesso.
Dato che console.iol è posizionato a fianco del servizio che esegue i test (nel container), andrà a sostituire quello dell'implementazione offerto di default da Jolie.

```
## console.iol ##
outputPort Console {
  RequestResponse:
    println( string )( void ),
    ...,
    ...
}
```

L'outputPort Console espone tutte le operation del console.iol di Jolie, e che verrà utilizzata dal servizio per mandare output.

```
embedded {
  Jolie:
    "./console-imp.ol" in Console
}
```

Viene incorporato (embedded) il servizio `console-imp.ol` all'interno della porta console.

console-impl.ol Questo servizio si occupa di implementare tutte le operation della console di Jolie ed esporle tramite una porta `LocalIn`. Quando `console-impl.ol` verrà incorporato renderà disponibile l'implementazione alternativa di console sulla porta `Console`.

```
## console-imp.iol ##
inputPort LocalIn {
Location: "local"
RequestResponse:
  println( string )( void ),
  ...,
  ...
}
```

In questo modo tutte le chiamate di stampa verranno tutte reindirizzate al di fuori del container tramite una outputPort `orchestratorOutput`

```
outputPort Orchestrator {
Protocol: sodep
RequestResponse:
  println( string )( void ),
  ...,
  ..
Location: ODEP_LOCATION
}
```

`ODEP_LOCATION` andrà a prendere a runtime il parametro passato al lancio del container che indica dove l'orchestrator esterno sarà in ascolto (ad esempio "socket://172.17.0.1:10005").

```
execution{ concurrent }
```

Per mantenere attivo il servizio che redireziona la console all'esterno.

```
main
{
  [println ( request )( response ){
    println@Orchestrator( request )( response )
  }]

  ...Altre operation di console..
}
```

Un esempio di come ogni `println` verrà inoltrato tramite la porta `Orchestrator` all'esterno del container.

9.3 Conclusioni

In questo capitolo è stato spiegato come funziona il tool JoUnit, che automatizza il testing unitario sfruttando una ambiente di deployment a container (Docker). JoUnit si inserisce in un progetto che raggruppa a supporto di un processo i Continuous Integration nel contesto dei microservizi.

Parte IV

Conclusioni

Capitolo 10

Contributi e Lavori Futuri

10.1 Contributi

I contributi apportati da questo lavoro sono:

- definizione degli step da seguire nei future works per sviluppare una serie di tool volti al supporto di un processo di continuous integration per architetture a microservizi
- progettazione e sviluppo del primo step del processo (Unit Test), che ha portato ad ottimizzare e migliorare l'orchestratore dei test (https://github.com/sanfo3855/JolieTestSuite) e la scrittura dei test case. Inoltre è stato anche sviluppato un software di installazione del tool di unit testing in locale (non su container) con una documentazione dettagliata consultabile al link https://github.com/sanfo3855/Installer-JTS
- sviluppo del tool JoUnit (https://github.com/jolie/jounit) per l'automazione del processo di unit testing, formato da generazione dei client e delle dipendenze, esecuzione del servizio da testare in un ambiente isolato anche dalle sue dipendenze, esecuzione dei test case scritti dallo sviluppatore.

10.2 Future Works

Come già anticipato nel capitolo 5 dopo aver sviluppato il tool per fare Unit Testing automatizzato ci sarà ancora del lavoro da fare. Riassumendo brevemente, i passi di sviluppo futuri

comprendono:

Integration Test Dopo che ogni microservizio ha passato il testing unitario, sarà necessario poter testare gruppi di microservizi che collaborano tra loro.
Possibili soluzioni potrebbero essere:

- sviluppo di un metodo per configurare e mettere in funzione un network docker composto da una serie di container collegati in modo che riescano a supportare l'esecuzione della rete di microservizi.
- raffinazione ulteriore dei servizi per fare unit testing, aggiungendo metodi per eseguire test su funzioni che dipendono dalla collaborazione tra più servizi e controllare i dati in ingresso e uscita tra i vai nodi di un gruppo di servizi

System Test Dopo aver testato gruppi di microservizi isolati, sarà necessario testare che ogni funzionalità si integri correttamente col sistema funzionante.
Sarà necessario trovare una modo per agganciare al sistema in esecuzione un gruppo di funzionalità nuove, in modo da poter testare se sono integrabili senza problemi e isolare queste funzionalità in modo che il sistema possa comunicarci solo per eseguire i test.

Deployment Hooks e GUI Passata la fase di testing (Unit, Integration e System) bisognerà fornire il deployment automatico di nuove funzionalità, magari coadiuvate da un'interfaccia grafica intuitiva a supporto di team di marketing o di management, in modo da rendere possibile anche per i non tecnici il monitoring dello stato e del deployment delle funzionalità offerte dei propri sistemi.

Capitolo 11

Ringraziamenti

Ringrazio il Professor. Davide Sangiorgi per aver svolto il ruolo da Relatore della mia tesi e per avermi permesso di svolgere questo importante progetto all'interno di un'azienda molto importante nel settore informatico.

Ringrazio il Dottor. Saverio Giallorenzo, il mio Correlatore, con il quale mi sono trovato sempre bene sia per il modo molto cordiale e amichevole di porsi nei miei confronti, sia per la disponibilità mostratami durante lo sviluppo di questo progetto e la scrittura della tesi, rispondendo a migliaia di mail e spendendo tempo con incontri di ore per definire parti non chiare. Va ringraziato anche per avermi fatto conoscere il linguaggio Jolie prima all'esame di Sistemi Operativi e poi nello sviluppo di questo progetto.

Ringrazio il Dottor. Claudio Guidi, mio referente presso Imola Informatica, l'azienda dove ho svolto il tirocinio, per avermi guidato nella realizzazione di questo primo tool. È stato molto interessante lavorare in un ambiente così affascinante e ricco di ispirazioni a un progetto così ambizioso che spero verrà portato col mio stesso interesse.

Ringrazio amici e amiche con cui ho condiviso gioie e dolori di questo percorso universitario triennale, passando nottate a programmare o giorni interi sui libri (sopratutto negli ultimi mesi), ma anche serate di svago e divertimento.

Infine ringrazio tantissimo i miei genitori per avermi sempre supportato (anche economicamente) in questi tre anni di studio e avermi permesso di andare a vivere e studiare in una città bellissima come Bologna.