Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Integrazione tra Reactive Programming e Microservizi in Jolie

Relatore:
Chiar.mo Prof.
Saverio Giallorenzo

Presentata da:
Aymen Tragha

IV Sessione
Anno Accademico 2021/2022

Introduzione

I microservizi sono un paradigma architetturale di sviluppo software in cui la logica dell'applicativo è distribuita tra una serie di unità che comunicano tra loro tramite API.
Jolie è un linguaggio di programmazione open source che offre varie primitive per definire i microservizi e gestirne il workflow.
Il reactive programming è un paradigma di programmazione che si basa su stream di dati asincroni e che intorno ad essi definisce il ciclo di vita del programma.
Dopo una panoramica su questi argomenti si discuterà su come implementare miscroservizi reactive in jolie e si fornirà un'implementazione dei concetti e delle primitive proposte.

Indice

Introduzione **i**

1 Microservizi **1**
1.1 Software monolitici 1
1.2 Caratteristiche 2
1.3 Vantaggi e svantaggi 3

2 Jolie **5**
2.1 Tipi di dato 5
2.2 Interfacce 6
2.3 Porte 7
2.3.1 Location 8
2.3.2 Protocol 8
2.3.3 Dynamic Binding 9
2.3.4 Chiamate locali 10
2.4 Servizi 10

3 Reactive Programming **13**

4 Progettazione di microservizi reactive **14**
4.1 Disquisizione 14
4.2 Event data structure 16
4.3 Pattern Observer 16
4.3.1 Descrizione 17

4.3.2 Request 18
4.3.3 ObserverInterface 18
4.3.4 ObservableInterface 18
4.3.5 SelfInterface 19
4.4 Event Propagation 20
4.4.1 Propagate 20
4.5 Esempio pratico 21
4.5.1 Observer 22
4.5.2 Observable 23
4.5.3 Esecuzione 28
4.6 Stato dell'arte e sviluppi futuri 29

Conclusioni **31**

Bibliografia **32**

Elenco delle figure

1.1 Esempio di un software monolitico 1
1.2 Esempio di architettura a microsevizi 2

4.1 Schema UML del pattern Observer 17
4.2 Ricezione delle subscribe, notifica e propagazione 28
4.3 Ricezione delle unsubscribe 28
4.4 propagazione senza notifica 29

Capitolo 1

Microservizi

I microservizi rappresentano un approccio architetturale in netto contrasto con i classici software monolitici.
In questo capitolo ci sarà una breve disamina dell'approccio monolitico[1] seguita da un'analisi dei tratti salienti dei microservizi[3, 15].

1.1 Software monolitici

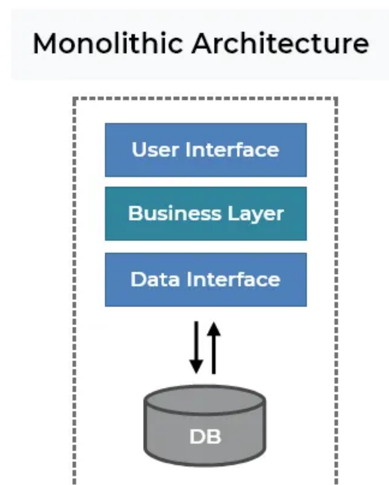

Figura 1.1: Esempio di un software monolitico

I software monolitici sono costituiti da un'unica grande unità che persiste i suoi dati in un unico database. Ci possono essere dei layer che creano una separazione logica grossolana(come avviene nell'esempio della figura 1.1, dove i layer vanno dal front-end(User interface) fino all'interfaccia con la base di dati) ma in ogni caso si parla di un sistema centralizzato.
Questo tipo di architettura ha il vantaggio di essere concettualmente semplice ma proprio per questo rende difficili l'aggiornamento, la scalabilità e il deployment del software, inoltre spesso complica o impedisce l'integrazione di nuove tecnologie.
In generale conviene usare questo approccio quando si ha a che fare con software relativamente piccoli o semplici.

1.2 Caratteristiche

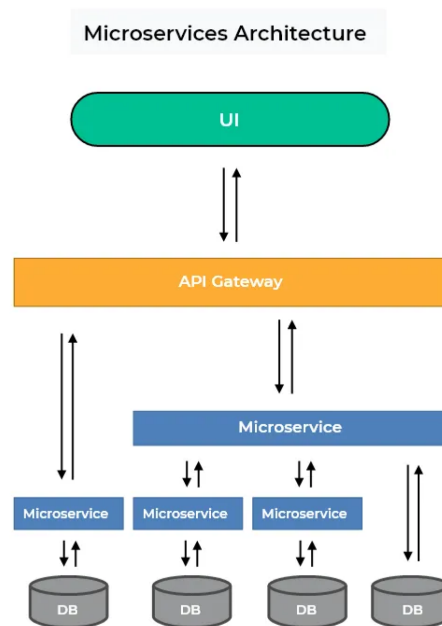

Figura 1.2: Esempio di architettura a microsevizi

I microservizi sono un sistema distribuito in cui ci sono varie unità(i servizi appunto) che comunicano in un nertwork.
I servizi hanno le seguenti caratteristiche:

- Comunicano tramite API.
- Sono loosely coupled, ovvero poco dipendenti tra di loro o indipendenti
- Sono bounded context, ogni servizio ha un proprio data model a sé stante.
- Ogni servizio si occupa di una porzione logicamente ben definita del sistema.
- Ogni servizio ha un proprio ambiente di persistenza.
- Team di sviluppatori diversi possono occuparsi di servizi diversi sinergicamente.
- Servizi diversi possono avere stack diversi di linguaggi, protocolli e tecnologie.

1.3 Vantaggi e svantaggi

Tra i vantaggi dei microservizi abbiamo:

- I servizi possono essere testati, deployati e aggiornati indipendentemente tra di loro.
- Si prestano a una metodologia di sviluppo agile grazie alla grande separazione logica che offrono.
- È più semplice contenere e isolare situazioni di fault, favorendo quindi la resilience del sistema.

Tra gli svantaggi invece abbiamo:

- Complessità architetturale e di management non necessariamente compensate dai vantaggi.
- Ogni servizio ha un proprio database e un proprio data model, di conseguenza possono esserci problemi di data consistency tra servizi che condividono dati.
  Un modo per arginare questo problema è adottare il principio dell'eventual consistency, per cui si definiscono meccanismi di reciproco aggiornamento tra servizi e ci si accontenta del fatto che prima o poi il model di un servizio sarà aggiornato o comunque non troppo datato.
- Se si ha a che fare con un sistema monolitico, il che non è improbabile considerando che essi sono stati i primi storicamente, può essere difficile migrare ai microservizi e non è detto che ne valga la pena.
- I microservizi dipendono dalla rete per la comunicazione, di conseguenza sono soggetti a problemi di network congestion e latency.

Capitolo 2

Jolie

Jolie è un linguaggio service-oriented progettato per offrire costrutti built-in per la creazione e la composizione di microservizi. In questo capitolo vedremo i costrutti più basilari.

2.1 Tipi di dato

In Jolie ci sono i seguenti tipi di dato[4]:

```
T ::= {void, bool, int, long, double, string, raw, any}
```

dove raw fa riferimento ai file binari e any indica che il tipo della variabile può essere uno qualunque dei precedenti, questo è possibile perché i tipi sono valutati a tempo di esecuzione dall'interprete(scritto in java) di Jolie, inoltre il tipo di dato contenuto in una variabile può cambiare durante il ciclo di vita del programma.
Oltre ai tipi base è possibile definire tipi complessi usando l'operatore . per creare dei sottonodi della radice, dove ogni nodo a sua volta ha un tipo base ed eventuali sottonodi.
Ecco un esempio di tipo di dato custom definito in questo modo:

```
type ShoppingList: void {
    fruits: int {
```

```
        bananas: int
        apples: int
    }
    notes: string
}
```

Possiamo definire istanze di ShoppingList in due modi:

1. Con l'operatore .

```
list.fruits = 5
list.fruits.bananas = 3
list.fruits.apples = 2
list.notes = "note"
```

2. Con l'operatore per la deep clone `<<`

```
list << {.fruits=5 .fruits.bananas=3
.fruits.apples=2, .notes="notes}
```

Inoltre è possibile accedere a una struttura dati dinamicamente in questo modo[5]:

```
list.fruits.("bananas") = 3
//equivalente a
list.fruits.bananas = 3
```

2.2 Interfacce

Um servizio comunica con altri servizi esponendo una o più interfacce e chiamando operazioni definite nelle interfacce degli altri servizi.
Un'interfaccia[6] ha la seguente struttura:

```
interface identifier {
    OneWay:
        operation1( message1 ),
        operation2( message2 ),
        //...,
         operationN( messageN )
```

```
    RequestResponse:
        operation1( request1 )( response2 ),
        operation2( request3 )( response4 ),
        //...
         operationN( requestrN )( responseN+1 )
}
```

Ci sono due tipi di operazioni:

- One Way, con cui il servizio riceve un messagio oppure ne manda uno asincronicamente.
- RequestResponse, con cui il servizio riceve una richiesta nel campo request e risponde al chiamante mettendo la risposta nel campo response, oppure al contrario manda sincronicamente una request e attende la risposta del destinatario che sarà poi disponibile nel campo response.

2.3 Porte

Le porte[7] permettono di impostare i parametri che regolamentano la comunicazione tra servizi e hanno la seguente struttura:

```
inputPort id {
    inputPort id {
    location: URI
    protocol: p
    interfaces: iface_1,
                ...,
                iface_n
}
outputPort id {
    location: URI
    protocol: p
    interfaces: iface_1,
                ...,
                iface_n
}
```

Si noti innanzitutto che ci sono due tipi di porte: le inputPort e le outputPort.
Esse hanno la stessa struttura ma due ruoli complementari in quanto, all'interno di un service, l'input port definisce un canale di ricezione di messaggi per quel service, mentre l'output port fa lo stesso ma per altri service, dunque un servizio userà le input port per definire i propri punti di accesso e le output port per definire quelli dei servizi a cui intende mandare messaggi.
Ogni porta ha un proprio nome identificativo e possono esserci molteplici porte per un singolo servizio.

2.3.1 Location

Il campo Location è un URI e indica appunto la locazione del servizio. Esso si compone di medium e indirizzo(con numero di porta).
Un esempio di location è:

```
1 socket://www.mysite.com:80/
```

Il medium è socket, l'indirizzo è la parte restante o e la porta è 80.
Al momento i media disponibili sono:

- local (Jolie in-memory communication);
- socket (TCP/IP sockets);
- btl2cap (Bluetooth L2CAP);
- rmi (Java RMI);
- localsocket (Unix local sockets).

2.3.2 Protocol

Il campo porotocol specifica il protocollo che si intende utilizzare.
I protocolli dipsonibili sono:

- http

- https
- soap
- sodep (protocollo specifico di Jolie)
- xmlrpc
- jsonrpc

2.3.3 Dynamic Binding

Location e protocol di un'output port possono essere modificati a runtime consentendo grande flessibilità.
Ecco un esempio in cui il dynamic binding[8] viene usato per mandare messaggi a destinatari diversi che usano lo stesso protocollo ed espongono la medesima operazione notify:

```
interface myInterface {
    OneWay:
        notify(message)
}
outputPort porta {
    location: "socket://localhost:8000"
    protocol: http
    interface: myInterface
}

//somewhere in the service logic
message="sup man"
foreach(receiver: receiverList) {
    porta.location = receiver
    notify@porta(message)
}
```

Si noti anche che un l'invio di un mesaggio ha il seguente schema:

```
Operation@Port(message)
```

2.3.4 Chiamate locali

Un servizio può chiamare sé stesso[9] nel seguente modo:

```
include "runtime.iol"

init {
    getLocalLocation@Runtime()( Self.location )
}
outputPort Self{
    Interfaces: LocalOperations
}

inputPort Self {
    Location: "local"
    Interfaces: LocalOperations
}
```

Oppure staticamente:

```
outputPort Self{
    Location: "local://Inner"
    Interfaces: LocalOperations
}

inputPort Self {
    Location: "local://Inner""
    Interfaces: LocalOperations
}
```

La location può essere definita come si vuole, l'importante è che venga utilizzato local come medium e che le location delle due porte siano coincidenti.

2.4 Servizi

Il service[10] è il costrutto fondamentale dei microservizi e di Jolie e il seguente esempio ne illustra la struttura:

```
interface MyServiceInterface {
    OneWay: f1(int)
```

```
    RequestResponse: f2 ( int )( int )
}

service MyService {

    execution: concurrent

    inputPort IP {
        location: "socket://localhost:8000"
        protocol: sodep
        interfaces: MyServiceInterface
    }

    init {
        //initialization stuff
    }

    main {
        [
        f2 ( number )( result ) {
            //do something
        }
        ]
        {
            //do something after response was sent
        }
        [ f1 ( number )]
        {
            //do something after message was received
        }
    }
}
```

Ci sono tre elementi degni nota:

1. Execution modality[11], ci sono tre modalità di esecuzione:

    - Single, vengono eseguiti init e main una volta sola, poi il processo muore.

- Sequential, il processo resta attivo e in ascolto e le chiamate ricevute vengono eseguite sequenzialmente.
- Concurrent, come sequential ma le chiamate ricevute sono eseguite in parallelo.

2. Init, un blocco che viene eseguito una volta sola durante lo start-up del service.
3. Main: Quando l'esecuzione è sequenziale o concorrente, nel main devono essere implementate le input choice, cha hanno il seguente schema:

```
[ IS_1 ] { branch_code_1 }
[ IS_i ] { branch_code_i }
[ IS_n ] { branch_code_n }

```

   Tra parentesi quadre si implementano le operazioni delle interfacce del servizio, tra quelle graffe invece si pongono eventuali azioni successive.

Ogni chiamata ricevuta ha un suo scope ma esiste anche uno scope globale grazie alla variabile global a cui è possibile aggiungere dei sottonodi ed è quindi possibile far sopravvivere una variabile locale oltre il suo scope di appartenenza.

Capitolo 3

Reactive Programming

Un programma è reactive nel momento in cui reagisce a eventi relativi al proprio stato interno o a eventi esterni.
Un evento può essere qualunque cosa avvenga durante il ciclo di vita del programma e spesso insieme a esso vengono emessi anche dei dati.
Un evento si può verificare più volte nel tempo generando uno stream di istanze di quell'evento e dei dati ad esso associati. Il mantra del reactive programming è "everything is a stream"[19].
Per avere un sistema performante é opportuno che la gestione degli eventi sia asincrona rispetto al flow del programma, perciò gli stream nel reactive programming sono asincroni.
Oltre alla possibilità di reagire agli eventi, è possibile definire relazioni tra di essi, per esempio si possono mappare due eventi in modo che l'emissione di uno comporti quella dell'altro.
Questo cambia in maniera significativa la formae mentis con cui lo sviluppatore concepisce il workflow del programma, poichè rispetto alla programmazione imperativa classica, con il reactive programming si ragiona ad un livello di astrazione più alto, si definiscono gli stream e le relazioni tra di essi e intorno a queste relazioni si costruisce la logica del programma.
A livello implementativo un design pattern fondamentale è l'observer pattern, che verrà discusso e implementato in 4.3

Capitolo 4

Progettazione di microservizi reactive

In questo capitolo si discuteranno e implementeranno i costrutti essenziali per avere un servizio reactive.

4.1 Disquisizione

Prima di vedere nei dettagli l'implementazione proposta, è utile soffermarsi sui principi, sulle idee e sui ragionamenti che ne costituiscono il fondamento.
L'obiettivo è di progettare delle primitive che permettano a un servizio Jolie di rispettare i requirements espressi nel capitolo 3, che possiamo riassumere brevemente come:

1. Emettere e ricevere eventi
2. Asincronicità
3. Definire relazioni tra stream di eventi e di dati

Prendendo ispirazione da librerie reactive preesistenti, in particolare da RxJS[18], appare evidente come il primo punto possa essere implementato efficacemente grazie al pattern observer.

Un service può essere un producer o un consumer di eventi e le due cose non si escludono a vicenda. Nel primo caso si parla di observer, che si mette in contatto con la sua controparte per essere notificato quando si verifica l'evento desiderato, nel secondo caso si parla di observable, che invece deve trasmettere gli eventi agli observer. La comunicazione tra i due cessa quando l'observer ne fa richiesta.
Per avere la massima flessibilità si cerca di fare in modo che un observable possa gestire più eventi e più observer contemporaneamente e allo stesso modo l'observer può richiedere più eventi con una sola richiesta e può collegarsi a più observable(con la premessa che a ogni servizio corrisponde un unico observable che gestisce un numero arbitrario di eventi). Per fare ciò si utilizzano varie strutture dati custom.
A questi ruoli corrispondono due interfacce che devono essere implementate seguendo le direttive descritte nelle sezioni successive. Non è stato fatto ricorso a costrutti più sofisticati(ad esempio embedding[12]) perché non è stato ritenuto necessario per gli obiettivi attuali.
La gestione degli eventi da parte dell'observable richiede l'utilizzo di global(vedi 2.4) perché le strutture dati legate agli eventi vanno mantenute oltre lo scope delle singole chiamate.
Il secondo punto può essere risolto osservando che le funzioni OneWay delle interfacce(vedi 2.2) permettono di stabilire una comunicazione tra servizi puramente asincrona(per avere una comunicazione bidirezionale asincrona è sufficiente avere per entrambi i servizi un'operazione di spedizione e una di ricezione[13]), perciò tutte le operazioni di tutte le interfacce saranno OneWay.
Infine, per avere un meccanismo generale per definire relazioni tra eventi, si osserva che prendendo in input un evento ed eseguendo un blocco di codice è possibile emettere un altro evento o al limite ottenere informazioni utili per la sua futura emissione.
Dunque, sebbene sia un po' macchinoso, è possibile modellare relazioni tra eventi definendo blocchi di codice che permettono a ogni evento di reagire all'emissione degli eventi da cui è dipendente. Inoltre ogni evento che ha

dipendenze dovrà avere una struttura dati dove immagazinare qualunque informazione utile, in modo da poter avere memoria del flusso di eventi emessi. Questo meccanismo è in sostanza propagazione di eventi che avviene all'interno di un servizio(observable o observer che sia) tramite una chiamata locale(vedi 2.3.4).
Un scenario semplice in cui sono presenti i concetti illustrati in questa sezione è reperibile a 4.5.

4.2 Event data structure

Il primo passo è individuare gli eventi che si intende gestire, dopodiché si definisce un tipo di dato evento in questo modo:

```
1 //? means that the subnode is optional
2 type Event {
3     name: string
4     data?: undefined
5     meta?: undefined    //for meta-info
6     subscribers?: undefined  //for notification
7     dependants?: undefined  //for propagation
8     store?: undefined   //for propagation
9 }
```

name nome univoco dell'evento

data dato eventualmente emesso insieme all'evento

meta campo opzionale per meta informazioni

Gli altri campi verranno trattati successivamente.

4.3 Pattern Observer

Il pattern observer[20] è uno dei building blocks di questa implementazione reactive. In questa sezione vedremo come funziona in generale e in che

modo ci torna utile.
È bene notare che nella parte implementativa verrà usata una terminologia leggermente diversa e più vicina a quella di librerie reactive come RxJS[18].

4.3.1 Descrizione

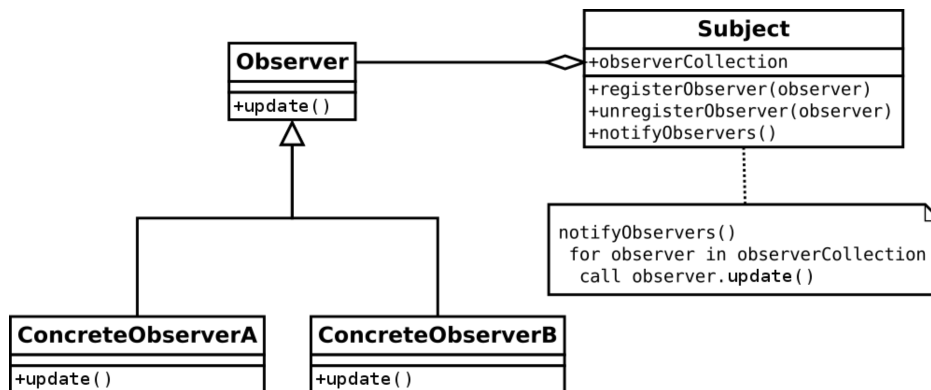

Figura 4.1: Schema UML del pattern Observer

Il design pattern Observer permette a un oggetto di aggiornare automaticamente una lista di altri oggetti quando si verificano cambiamenti nel suo stato interno. Il funzionamento è il seguente:

1. L'observer si registra presso il subject chiamandone il metodo registerObserver.

2. L'observer viene aggiunto all'observerCollection del subject.

3. Quando si verifica un cambiamento di stato nel subject, l'observerCollection viene ciclata e di ogni observer viene chiamato il metodo update, che implementa la reazione al cambiamento.

4. Quando l'observer non è più interessato allo stato del subject, chiama unregisterObserver e viene rimosso dall'observerCollection.

4.3.2 Request

Tramite il pattern observer è possibile avere un meccanismo per la trasmissione e la ricezione di eventi tra servizi.
Innanzitutto si definisce un tipo di dato request:

```
//sub or unsub request
type Request {
    loc: string //observer location
    events: undefined //tree where subnodes are events
    options?: undefined
}
```

Con questa struttura abbiamo:

loc sringa che indica la location dell'observer.

events struttura che ha come sottonodi gli eventi a cui ci si vuole iscrivere.

options parametro opzionale per settare eventuali paramteri di richiesta.

4.3.3 ObserverInterface

L'observer implementa la seguente interfaccia:

```
interface ObserverInterface {
    OneWay:
        confirm(bool),  //confirm subscription
        notify(Event)
}
```

L'ObserverInterface espone le operazioni:

confirm per ricevere la conferma della subscription.

notify per agire una volta ricevuto l'evento.

4.3.4 ObservableInterface

L'observervable implementa la seguente interfaccia:

```
interface ObservableInterface {
    OneWay:
        subscribe(Request),
        unsubscribe(Request)
}
```

L'ObservableInterface espone le operazioni:

subscribe prende la subscription request e itera gli eventi ivi presenti e per ogni evento accede alla sua subscriber list e aggiunge la locazione del mittente. Poi chiama la confirm dell'observer mittente.

unsubscribe rimuove il mittente dalle subscriber list degli eventi specificati nella request.

4.3.5 SelfInterface

Observer diversi possono iscriversi allo stesso evento, quindi è necessario, nel momento della sua emissione, iterare la lista dei suoi subscriber e mandare l'evento a ciascuno di loro, per fare ciò si usa una porta Self(vedi 2.3.4) con la seguente interfaccia:

```
interface SelfInterface {
    OneWay:
        emit(Event),    //wrapper for the other two
        propagate(Event),
        notifyAll(Event)
}
```

emit wrapper che chiama notifyAll e poi propagate, è la funzione che viene usata per lanciare eventi.

propagate per la propagazione di eventi vedi 4.4.1.

notifyAll notifica un evento a tutti i subscriber interessati.

4.4 Event Propagation

Fino ad ora abbiamo lasciato allo sviluppatore la totale responsabilità di implementare la logica degli eventi e finché questi sono indipendenti tra di loro, il problema non si pone, tuttavia gli eventi possono avere relazioni più o meno complesse e quindi serve un meccanismo generale per gestire le dipendenze tra eventi.

4.4.1 Propagate

Quando un evento si verifica, si chiama emit@Self. Essa notifica l'evento ai subscriber con la notifyAll e poi lo propaga con propagate.
Prima di vedere lo schema di propagate, chiariamo alcuni aspetti:

- Un evento evento può dipendere da altri eventi o avere eventi che dipendono da lui.
- Ogni evento ha una lista di eventi che dipendono da lui. È compito dello sviluppatore stabilire i rapporti di dipendenza tra gli eventi e riempire coerentemente le liste di dependants.
- Un evento che ha almeno una dipendenza ha a disposizione uno store(sottonodo della struttura dati Event, vedi 4.2) in cui salvare qualsivoglia informazione utile per la sua logica.
- All'interno di propagate si associa un blocco di codice a ogni coppia {A,B} con A dipendente da B, in questo modo A può agire sapendo che B é avvenuto.

```
1 propagate(Event) {
2     foreach(dependant : Event.dependants) {
3         //assuming events A1...An have at least one
    dependency each
4         if (dependant == A1) {
5             //assuming A1 depends on events B1...Bn
6             if(Event == B1) {
```

```
                //do something
                //store some info
                dependant.store.info = info
            }
            ...
            if(Event == Bn) {
                ...
            }
        }
        ...
        if (dependant == An) {
        ...
        }
    }
}
```

In questo pseudocodice vediamo che all'interno di propagate vengono iterati gli eventi che dipendono dall'evento propagato.
All'interno del foreach c'è un blocco if in cui si controlla chi sia il dependant corrente, poi vi è un if annidato che racchiude il codice da eseguire in seguito all'evento appena emesso, in questo modo, una volta defniti dei blocchi per ogni coppia dependant-dependency, si permette al dependant di reagire all'emissione della dependency.
Infine l'accesso allo store del dependant permette di salvare informazioni utili per la sua logica.

4.5 Esempio pratico

In questa sezione vediamo un semplice esempio in cui vengono utilizzate le primitive discusse in questo capitolo.
Ci sono tre servizi, due observer e un observable. Quest'ultimo gestisce tre eventi A,B e C che emettono le stringhe "Alice", "Bob" e "Callisto" rispettivamente.
A e B sono indipendenti tra di loro mentre C dipende da entrambi, in parti-

colare C è emesso se e solo se, a partire dall'ultima emissione di C, sia A che B sono già stati emessi almeno una volta.
In seguito a una subscribe per A(o per B), l'observable comincia a emettere A(o B) ogni cinque secondi per venti volte. C invece non viene emesso direttamente, bensì è generato mediante propagazione.
Un observer si iscrive per A e C, mentre l'altro si iscrive per B e C ed entrambi, quando hanno raccolto dieci istanze di un evento, chiamano l'unsubscribe per quell'evento.
IL codice completo e le istruzioni per eseguirlo sono disponibili qui.
Nelle prossime sezioni invece verranno mostrati i tratti salienti.

4.5.1 Observer

Questo è il codice dell'observer che si iscrive ad A e C, l'altro observer è identico se non per B al posto di A e per la location dell'inputPort diversa.

Inizializzazione:

```
init {
    with(subRequest) {
        .loc="socket://localhost:8001"
        with(.events) {
            .A="A"
            .C="C"
        }
    }
    subscribe@Observable(subRequest)
    global.countA = 0
    global.countC = 0
}
```

Nel blocco init si inizializza l'observer facendo la subscribe agli eventi A e C, poi si inizializzano i counter dei due eventi a 0, si usa global(vedi 2.4) in modo da poter tenere il conto tra una notify e l'altra.
with è un costrutto che fa da shortcut e permette di non ripetere ogni volta

il nome della radice quando si accede ai sottonodi.

Notify:

```
[
    notify(Event)
]
{
    if(Event.name == "A") {
        print@Console( "received event A with value "
        +Event.data+"\n")( )
        global.countA = global.countA +1
        if(global.countA == 10) {
            unsubscribe@Observable({
            .loc="socket://localhost:8001"
            .events.A="A"})
        }
    }
    if(Event.name == "C") {
        print@Console( "received event C with value "
        +Event.data+"\n" )( )
        global.countC = global.countC + 1
        if(global.countC == 10) {
            unsubscribe@Observable({
            .loc="socket://localhost:8001"
            .events.C="C"})
        }
    }
 }
```

Quando viene ricevuto un evento viene incrementato il relativo counter, se quest'ultimo arriva a dieci allora si effettua l'unsubscribe per quell'evento.

4.5.2 Observable

Questo è il codice dell'observable:

Inizializzazione:

```
with(global.events) {
            with(.A) {
                .name= "A"
                .data = undefined
                with(.dependants) {
                    .C = undefined
                }
            }
            with(.B) {
                .name= "B"
                .data = undefined
                with(.dependants) {
                    .C= undefined
                }
            }
            with(.C) {
                .name= "c"
                .data = undefined
                .store = undefined
            }
        }
```

L'inizializzazione dell'observable consiste nella creazione delle strutture dati associate agli eventi A,B e C e alla loro aggiunta al sottonodo events di global.
A e B hanno C come dependant, mentre C ha lo store in evidenza e privo di contenuti al momento.

Subscribe:

```
[
    subscribe(subReq)
]
{
    print@Console( "received subscription
    request to events:\n" )(  )
    subLocation = subReq.loc
    foreach( eventName : subReq.events ) {
```

```
9          println@Console( "-"+eventName )()
10         with(global.events) {
11             .(eventName).subscribers.(subLocation)
12             = undefined
13         }
14     }
15     Observer.location = subLocation
16     confirm@Observer(true)
17     Observer.location = "placeholder"
18     if(subLocation=="socket://localhost:8001")
19         ev << {.name="A", .data="Alice"}
20     else if(subLocation=="socket://localhost:8002")
21         ev << {.name="B", .data="Bob"}
22     for(i=1, i<=20, i++) {
23         sleep@Time( 5000 )()
24         print@Console( "\n" )( )
25         emit@Self(ev)
26     }
27 }
```

In primo luogo viene iterato il nodo events della subscription request e ad ogni evento della richiesta viene aggiunto il subscriber(identificato dalla sua location) al nodo subscribers.
Qui termina la subscribe in senso stretto, quello che viene dopo è la logica di emissione degli eventi A e B. Ogni cinque secondi, per venti volte, viene emesso un evento A o B(a seconda di ciò a cui si è iscritto il subscriber).
L'emit ha la seguente implementazione:

```
1 [
2     emit(Event)
3 ]
4 {
5     notifyAll@Self(Event)
6     propagate@Self(Event)
7 }
```

Unsubscribe:

```
[
    unsubscribe(unsubReq)
]
{
    subLocation = unsubReq.loc
    if(subLocation=="socket://localhost:8001")
        obs = "ObserverA"
    else if(subLocation==
    "socket://localhost:8002")
        obs = "ObserverB"
    println@Console( "received unsubscribe request from "
    +obs+" to events:" )( )
    foreach( eventName : unsubReq.events ) {
        println@Console( "-"+eventName+"\n" )()
        with(global.events) {
            undef(.(eventName).subscribers.(subLocation))
        }
    }
}
```

L'unsubscribe itera il nodo events della richiesta e per ogni evento ivi presente rimuove l'ex subscriber dal nodo subscribers tramite la primitiva undef, che rende undefined la variabile che prende in input.

NotifyAll:

```
[
    notifyAll(Event)
]
{
    foreach(sub : global.events.(Event.name).subscribers ) {
        Observer.location = sub
        if(sub=="socket://localhost:8001")
            obs = "ObserverA"
        else if(sub=="socket://localhost:8002")
            obs = "ObserverB"
        println@Console( "notifying event "+Event.name+" to
   "+obs )( )
```

```
12          notify@Observer(Event)
13      }
14      Observer.location = "placeholder"
15  }
```

Sfruttando il dynamic binding(vedi 2.3.3) si itera tra i subscriber dell'evento emesso e per ognuno di essi si chiama la notify.

Propagate:

```
1   [
2       propagate(Event)
3   ]
4   {
5       print@Console( "propagating "+Event.name+"\n" )( )
6       foreach(dependant : global.events.(Event.name).dependants
        ) {
7           if (dependant == "C") {
8               if(Event.name == "A") {
9                   global.events.("C").store.A=true
10              }
11              if(Event.name == "B") {
12                  global.events.("C").store.B=true
13              }
14              if(global.events.("C").store.A &&
                   global.events.("C").store.B) {
15                  global.events.("C").store.A = false
16                  global.events.("C").store.B = false
17                  notifyAll@Self({.name="C" .data="Callisto"})
18                  propagate@Self({.name="C .data="Callisto"})
19              }
20          }
21      }
22  }
```

Seguento lo schema proposto in 4.4.1, l'evento C ha le dipendenze A e B e per ognuna di esse è definito un blocco in cui si accede allo store per segnarsi che l'evento propagato.

Infine nell'ultimo if si controlla se sono verificate le condizioni per emettere C, in caso affermativo lo si emette e si resetta lo store.

4.5.3 Esecuzione

```
activeMS\test>jolie ObserverA.ol
subscription confirmed
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
```

```
activeMS\test>jolie Observable.ol
received subscription request to events:
-A
-C
received subscription request to events:
-B
-C

notifying event A to ObserverA
propagating A

notifying event B to ObserverB
propagating B
propagating C
notifying event C to ObserverB
notifying event C to ObserverA

notifying event A to ObserverA
propagating A

notifying event B to ObserverB
propagating B
notifying event C to ObserverB
notifying event C to ObserverA
propagating C
```

```
eactiveMS\test>jolie ObserverB.ol
subscription confirmed
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event C with value Callisto
received event B with value Bob
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
```

Figura 4.2: Ricezione delle subscribe, notifica e propagazione

L'observable riceve dall'observerA una subscribe request per gli eventi A e C, mentre da observerB riceve una richiesta per B e C. Gli eventi A e B vengono emessi, notificati e propagati e lo stesso avviene per C quando le sue condizioni sono verificate.

Intanto i due observer continuano ad accumulare i rispettivi eventi.

```
activeMS\test>jolie observerA.ol
subscription confirmed
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
```

```
notifying event B to ObserverB
propagating B
received unsubscribe request from Observer
B to events:
-B


notifying event A to ObserverA
propagating A
notifying event C to ObserverB
propagating C
received unsubscribe request from Observer
A to events:
notifying event C to ObserverA
-A

received unsubscribe request from Observer
B to events:
-C

received unsubscribe request from Observer
A to events:
-C
```

```
eactiveMS\test>jolie ObserverB.ol
subscription confirmed
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
```

Figura 4.3: Ricezione delle unsubscribe

Gli observer hanno raggiunto il tetto massimo per per i rispettivi eventi e

quindi chiamano unsubscribe.

```
activeMS\test>jolie observerA.ol
subscription confirmed
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
received event A with value Alice
received event C with value Callisto
```

```
received unsubscribe request from Observer
A to events:
-C

propagating B

propagating A
propagating C

propagating B

propagating A
propagating C

propagating B

propagating A
propagating C

propagating B

propagating A
propagating C
```

```
eactiveMS\test>jolie ObserverB.ol
subscription confirmed
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
received event B with value Bob
received event C with value Callisto
```

Figura 4.4: propagazione senza notifica

Concluse le unsubscribe, l'observable per un paio di iterazioni continua ad emettere A,B e C, che vengono propagati ma non notificati in quanto non ci sono più subscriber a questo punto dell'esecuzione.

4.6 Stato dell'arte e sviluppi futuri

L'implementazione reactive proposta in questo elaborato permette a un servizio Jolie di mandare e ricevere eventi ad altri servizi asincronicamente e quindi di gestire stream di eventi e di dati, inoltre il meccanismo di propagazione permette di definire relazioni e dipendenze tra eventi e quindi in definitiva permette allo sviluppatore di pensare in modo reactive.
A partire da ciò ci vari sviluppi possibili, alcuni di questi sono:

- Implementazione del pattern publisher/subscriber[16] per una comunicazione tra servizi mediata da un message broker.
- Utilizzo dell'event sourcing[2] per rappresentare l'evoluzione dello stato interno di un service come stream di eventi.
- Implementazione del design pattern CQRS[14] per gestire l'accesso al database, questo pattern inoltre è sinergico con l'event sourcing.

- Utilizzo di una queue[17] che faccia da buffer di eventi per gestire carichi variabili di dati in ingresso.
- Incrementare la resilience del sistema tramite meccanismi di fault handling e recovery.

Conclusioni

È stata fornita una breve trattazione sui microservizi, sui costrutti e sulle caratteristiche basilari di Jolie e del reactive programming.
Poi sono state discusse, definite e implementate le primitive essenziali per applicare un approccio reactive a un servizio Jolie ed è stato mostrato un semplice esempio di emissione, ricezione e propagazione di eventi tra servizi.
Questi costrutti sono i blocchi di base da cui è possibile partire per implementare concetti e design pattern più avanzati del reactive programming.

Bibliografia

[1] Henrique Siebert Domareski. *Monolithic & Microservices Architecture*. URL: https://henriquesd.medium.com/monolithic-microservices-architecture-239e8799d3e1. (ultimo accesso: 01.03.2023).

[2] Martin Fowler. *Event Sourcing*. URL: https://martinfowler.com/eaaDev/EventSourcing.html. (ultimo accesso: 04.03.2023).

[3] IBM. *What are microservices?* URL: https://www.ibm.com/topics/microservices. (ultimo accesso: 01.03.2023).

[4] Jolie. *Jolie documentation*. URL: https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/basics/data-types/index.html. (ultimo accesso: 01.03.2023).

[5] Jolie. *Jolie documentation*. URL: https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/basics/data-structures/index.html. (ultimo accesso: 01.03.2023).

[6] Jolie. *Jolie documentation*. URL: https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/basics/interfaces/index.html. (ultimo accesso: 01.03.2023).

[7] Jolie. *Jolie documentation*. URL: https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/basics/ports/index.html. (ultimo accesso: 01.03.2023).

[8] Jolie. *Jolie documentation*. URL: https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/basics/dynamic-binding/index.html. (ultimo accesso: 01.03.2023).

[9] Jolie. *Jolie documentation*. URL: https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/locations/local/index.html. (ultimo accesso: 01.03.2023).

[10] Jolie. *Jolie documentation*. URL: https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/basics/services/index.html. (ultimo accesso: 01.03.2023).

[11] Jolie. *Jolie documentation*. URL: https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/basics/processes-and-sessions/processes/index.html. (ultimo accesso: 01.03.2023).

[12] Jolie. *Jolie documentation*. URL: https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/architectural-composition/embedding/index.html. (ultimo accesso: 01.03.2023).

[13] Jolie. *Jolie documentation*. URL: https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/architectural-patterns/synchronous-vs-asynchronous/index.html. (ultimo accesso: 01.03.2023).

[14] Microsoft. *CQRS pattern*. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs. (ultimo accesso: 04.03.2023).

[15] Microsoft. *Microservice architecture style*. URL: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices. (ultimo accesso: 01.03.2023).

[16] Microsoft. *Publisher-Subscriber pattern*. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber. (ultimo accesso: 04.03.2023).

[17] Microsoft. *Queue-Based Load Leveling pattern*. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling. (ultimo accesso: 04.03.2023).

[18] RxJS. *RxJS*. URL: rxjs.dev/guide/overview. (ultimo accesso: 01.03.2023).

[19] André Staltz. *The introduction to Reactive Programming you've been missing*. URL: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754. (ultimo accesso: 01.03.2023).

[20] Wikipedia. *Observer pattern*. URL: https://en.wikipedia.org/wiki/Observer_pattern. (ultimo accesso: 01.03.2023).