ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

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

IL LINGUAGGIO BALLERINA

Elaborato in
SISTEMI EMBEDDED E INTERNET OF THINGS

Relatore
Prof. ALESSANDRO RICCI

Presentata da
MARCO FRATTAROLA

Anno Accademico 2021 – 2022

Alla mia famiglia

Indice

Introduzione

Il linguaggio Ballerina è stato sviluppato dall'azienda WSO2 [1], che si occupa, principalmente, dello sviluppo di API. WSO2 ha sviluppato Ballerina con lo scopo di trovare un modo più semplice ed efficacie per sviluppare e integrare microservizi. Ballerina è un linguaggio open-source, cloud-native, progettato, quindi, con l'intento di alleggerire il carico dello sviluppo e dell'integrazione associato alle applicazioni aziendali. Questo linguaggio semplifica il modo in cui un programma comunica con la rete, che di solito avviene tramite un API (Application Program Interface). Ballerina tenta di creare un **sistema integrato**, riunendo i concetti, le idee e gli strumenti essenziali di integrazione di un sistema distribuito e offrendo un ambiente concorrente e sicuro per supportare lo sviluppo di API. Un ambiente sicuro è quello in cui una certa parte di codice può accedere solo alle posizioni di memoria a cui è programmato per accedere. Secondo Tyler Jewell, CEO di WSO2, oltre il 50% del tempo e dei costi per la trasformazione digitale e per i progetti API, all'interno delle imprese, è l'**integrazione** [2].

Il linguaggio Ballerina è stato creato, quindi, per **semplificare** l'integrazione di servizi, sistemi e dati disparati, in modo che gli sviluppatori possano dedicare più tempo a concentrarsi sulle funzionalità. Il tipo di ambiente concorrente e sicuro offerto da Ballerina, consente l'implementazione di microservizi con funzionalità come transazioni, messaggistica affidabile, workflow ed elaborazione di stream.

Ballerina è un linguaggio di programmazione orientato al **cloud computing** ed è stato progettato attorno ai seguenti principi fondamentali:

- **Diagrammi di sequenza**: Ogni programma Ballerina può essere rappresentato tramite un diagramma di sequenza. In ogni diagramma vengono rappresentate, in modo grafico, le interazioni tra i vari attori presenti, automatizzando il sistema di documentazione dei sistemi interconnessi.

- **Osservabilità**: Il linguaggio supporta le metriche basate su microservizi, tracing e logging, senza la necessità di aggiungere delle librerie aggiuntive. Il binario prodotto da Ballerina genera automaticamente i dati.

Per ottenere queste funzionalità, gli utenti devono semplicemente configurare i servizi e i tools utili ad analizzare e mostrare i dati generati.

- **Workers concorrenti**: Il modello di esecuzione del linguaggio è composto da delle leggere **unità di esecuzione parallela**, note come workers. I workers operano seguendo una politica completamente non bloccante. Questa politica rende i workers degli attori concorrenti **indipendenti**, che possono interagire tra loro tramite lo scambio di messaggi.

- **Orientato alla rete**: Normalmente, i sistemi di rete restituiscono messaggi con differenti tipi di payload ed errori. Ballerina ha un *type system* strutturale, con primitive, oggetti, unioni di più tipi e tuple. Questo type system abbraccia questa variabilità e semplifica l'interazione di rete.

- **Supporto integrato per i container**: Ballerina consente di specificare delle annotazioni nel codice, in modo che il compilatore crei istantaneamente un Dockerfile [3] e lo accoppi con un'immagine, che può quindi essere eseguita come contenitore. Viene offerta anche una libreria per il supporto di Kubernetes [4].

Molti linguaggi di programmazione e servizi di integrazione, utilizzati oggi dagli sviluppatori, incorporano la logica di programmazione all'interno di file YAML [5], XML [6] o altri tipi di file. Ballerina consente agli sviluppatori di scrivere **codice** per integrare gli elementi, piuttosto che utilizzare complessi schemi di integrazione. Il type system di Ballerina rende tabelle JSON [7] o XML, record, mappe ed errori, delle **primitive**, eliminando la necessità di utilizzare delle librerie per lavorare con queste strutture dati fondamentali. Di conseguenza, gli sviluppatori possono effettuare manipolazioni su questi tipi di strutture dati utilizzando semplici costrutti all'interno del codice sorgente.

Il titolo "Ballerina" deriva dall'ammirazione del CEO e fondatore della WSO2, Sanjiva Weerawarana, per quanto siano strettamente coordinati i balletti e per come questa coordinazione crei un flusso aggraziato.

Nei seguenti capitoli verranno, inizialmente, illustrate le principali caratteristiche del linguaggio. Successivamente, verranno trattate le funzionalità che più caratterizzano Ballerina ed infine verrà mostrato un caso di studio, che mette in evidenza i vantaggi che il linguaggio offre nei suoi contesti di applicazione.

Capitolo 1

Caratteristiche principali

1.1 Funzionamento

In quanto linguaggio di programmazione concorrente, compilato, type safe, il codice di Ballerina viene raccolto in **bytecode** e successivamente eseguito in una macchina virtuale (VM). All'inizio del suo sviluppo, il team di Ballerina ha tentato di implementare la propria macchina virtuale, ma ha riscontrato colli di bottiglia nelle prestazioni. Conosciuta come Ballerina Virtual Machine (BVM) [8], questa VM eseguiva i programmi Ballerina interpretando il bytecode BVM emesso dal compilatore Ballerina. Tuttavia, il team di Ballerina alla fine ha deciso che BVM, implementato in Java [9], non era pronto per l'uso e ha deciso di includere un compilatore destinato a JVM [10], con il rilascio della versione 1.0 [11].

Ballerina non è né un linguaggio orientato agli oggetti (OOP), né funzionale. È possibile definirlo come un linguaggio **data-oriented**. Sebbene il linguaggio abbia gli oggetti, i metodi di sviluppo non sono completamente orientati agli oggetti. I principi di progettazione utilizzano una sequenza che rappresenta dei concetti, inclusi elementi dichiarativi, che non sono puramente OOP. Sia gli oggetti che le funzioni sono, comunque, dei concetti di prima classe, dando agli sviluppatori la possibilità di scegliere l'opzione migliore in base alle loro esigenze.

1.2 Sintassi

La sintassi generale di un programma Ballerina tende ad essere simile al linguaggio C [12]. I commenti possono essere inseriti con la notazione *"//"*. Le definizioni e le dichiarazioni dei moduli utilizzano parentesi graffe e terminano con il punto e virgola [13].

È possibile utilizzare caratteri Unicode negli identificatori sia in maniera esplicita, sia utilizzando la sintassi "\u{H}", dove H è il **code point** (il valore numerico che identifica il carattere) in esadecimale del carattere Unicode [14].

Alcune parole chiave come *int*, *function*, *string*, sono riservate. Tuttavia è possibile utilizzarle come identificatori inserendo una virgoletta come prefisso.

```
import ballerina/io;

// È possibile utilizzare identificatori Unicode.
function štãmpa(string çittà) {
    // \u{H} per specificare un carattere usando
    // il code point Unicode in esadecimale.
   io:println(çitt\u{0E0}); //  -> \u{0E0} = à
}

string 'string = "abc";
```

Aspetti di sintassi più specifici verranno illustrati nelle prossime sezioni.

1.3 Struttura dei programmi

1.3.1 Moduli

Un programma Ballerina è diviso in diversi **moduli** [15]. Ogni modulo ha una forma **Sorgente** ed una **Binaria**. La forma sorgente di un modulo consiste in una collezione ordinata di una o più parti sorgente. Ognuna di queste parti è una sequenza di bytes che sono la codifica in UTF-8 di parte del codice sorgente del modulo.

Un modulo sorgente può referenziare altri moduli e può essere compilato separatamente in un **modulo binario**. La compilazione di un sorgente, infatti, richiede accesso soltanto alla forma binaria dei moduli a cui esso fa riferimento.

Un modulo sorgente identifica ogni modulo a cui esso fa riferimento nel seguente modo:

nome dell'organizzazione + nome del modulo

Il nome del modulo, a sua volta, può essere diviso in una o più parti. Sia il nome dell'organizzazione che delle singole parti del nome del modulo sono stringhe Unicode. La stringa *ballerina* come nome dell'organizzazione è riservata all'utilizzo della piattaforma.

```
import ballerina/io;

public function main() {
    io:println("x");
}
```

Le gerarchie dei moduli seguono la notazione *org/x.y.z*. In questo esempio, l'organizzazione è *ballerina* e il modulo è *io*. L'identificatore associato che fa riferimento a questo modulo della libreria è impostato sul nome del modulo *io*. Per sovrascrivere l'associamento dell'identificatore predefinito per il modulo, è possibile usare la parola chiave *as* come segue:

```
import <nome organizzazione>/<nome modulo> as <identificatore>.
```

La funzione *main* è il punto di ingresso del programma e la parola chiave **public** rende questa funzione visibile all'esterno del modulo. La notazione *"z:f"* punta a un simbolo f (in questo caso una funzione) definito all'interno del modulo *x.y.z*.

1.3.2 Packages

La piattaforma di Ballerina utilizza un sistema di **packaging** dei moduli, che permette a più moduli di essere combinati in un solo package. Il sistema di packaging considera la prima parte di ogni nome del modulo come **nome del package**. Tutti i moduli nello stesso package condividono, quindi, lo stesso nome del package.

Ogni package, a sua volta, ha una forma sorgente ed una binaria:

- La forma sorgente raccoglie tutte le forme sorgenti dei moduli all'interno del package utilizzando un filesystem gerarchico.
- La forma binaria raccoglie tutte le forme binarie dei moduli all'interno del package come sequenza di bytes.

Ai package binari può essere assegnata una versione e possono essere raccolti in un **repository**. Un singolo repository può contenere più versioni dello stesso package e per questo, al suo interno, i package vengono organizzati in tre livelli gerarchici:

1. Organizzazione;
2. Nome del package;
3. Versione.

Il formato sorgente di un package include, inoltre, un file *Ballerina.toml*, che permette di gestire le versioni dei package usati, per i moduli che vengono referenziati. Il sistema di packaging permette, infine, di gestire quali moduli vengono **esportati** da un package: i moduli che non vengono esportati da un package sono visibili soltanto all'interno del package stesso.

1.4 Valori, tipi e variabili

Il type system **flessibile** di Ballerina aiuta gli sviluppatori a lavorare con le risorse di rete. In un linguaggio di programmazione, il type system è la base per rappresentare i dati ed implementare la logica del programma. Mentre alcuni linguaggi forniscono funzionalità di base, altri cercano di fornire funzionalità integrate per alcuni domini specializzati.

Con l'aumentare dei servizi disponibili nel cloud, il dominio della programmazione **distribuita** in rete è cresciuto. Considerato che, allo sviluppatore viene data la responsabilità aggiuntiva di lavorare con le risorse di rete nel proprio codice, il linguaggio di programmazione stesso deve essere d'aiuto in questa operazione. Ecco perché il type system **network-friendly** di Ballerina è specializzato per questo dominio.

1.4.1 Tipi fondamentali

Tutti i valori all'interno di un programma Ballerina appartengono ad uno ed un solo **tipo elementare**. Esistono quattro tipi fondamentali di valori, ognuno dei quali corrisponde ad un tipo elementare:

- **Valori semplici**: Quei valori che non sono costruiti a partire da altri valori (ad esempio booleani o numeri floating points).
- **Valori strutturati**: Quei valori che contengono altri valori al loro interno (ad esempio mappe o liste);
- **Valori sequenziali**: Quei valori che combinano aspetti di valori semplici e valori strutturati;
- **Valori Comportamentali**: Quei valori che non sono soltanto dati (ad esempio funzioni o oggetti).

1.4.2 Identità di archiviazione

In ballerina esiste una distinzione fondamentale tra i valori che hanno una **Identità di archiviazione** e quelli che non ne hanno. Un valore che possiede un'identità di archiviazione ha un'identità che deriva dalla locazione di

memoria dove il valore viene memorizzato. Tutti i valori strutturali e comportamentali hanno un'identità di archiviazione mentre, invece, tutti i valori semplici non la posseggono. L'identità di archiviazione dei valori sequenziali verrà approfondita nella sezione 1.4.8.

I valori possono essere memorizzati in **variabili**, come **membri di strutture** o come **costituenti di sequenze**. Tuttavia, quando un valore ha un'identità di archiviazione quello che viene realmente memorizzato all'interno della variabile, struttura o sequenza è un **riferimento** alla locazione di memoria dove il valore è memorizzato. Per quanto riguarda i valori che non hanno un'identità di archiviazione, invece, viene memorizzato il valore stesso.

Per valori con un'identità di archiviazione esistono i seguenti concetti:

- **Creare un nuovo valore**: Consiste nel creare un valore con un'identità di archiviazione che è diversa da qualsiasi altro valore esistente.
- **Copiare un valore**: Consiste nel creare un valore identico ad un altro valore già esistente, fatta eccezione per la sua identità di archiviazione.
- **Confrontare due valori**: Ballerina permette di verificare se due valori hanno la stessa identità di archiviazione, ma non espone le specifiche locazioni di memoria dei valori.

L'identità di archiviazione in Ballerina permette di rappresentare i valori non solo come alberi, ma anche come **grafi**.

1.4.3 Shapes

I programmi Ballerina usano i **tipi** per categorizzare i valori sia in fase di compilazione che in fase di esecuzione. I *tipi* trattano un'astrazione dei valori che non considerano l'identità di archiviazione. Questa astrazione viene chiamata **shape**.

Un tipo, quindi, denota un insieme di **shapes**. È importante evidenziare che ogni valore ha una sua shape corrispondente ed è specifica rispetto al suo **tipo elementare**: se due valori hanno due diversi tipi elementari, allora essi hanno due shapes diverse. L'insieme delle shapes dei valori che compongono un valore strutturato sono parte della shape del valore strutturato stesso.

```
type DoorState record {|
    boolean open;
    boolean locked;
|};
```

Se consideriamo questo esempio, il tipo DoorState è un **record** con due valori booleani, che rappresenta lo stato di una porta. Tutte le possibili shapes di questo tipo saranno tutte le combinazioni di valori che esso può assumere.

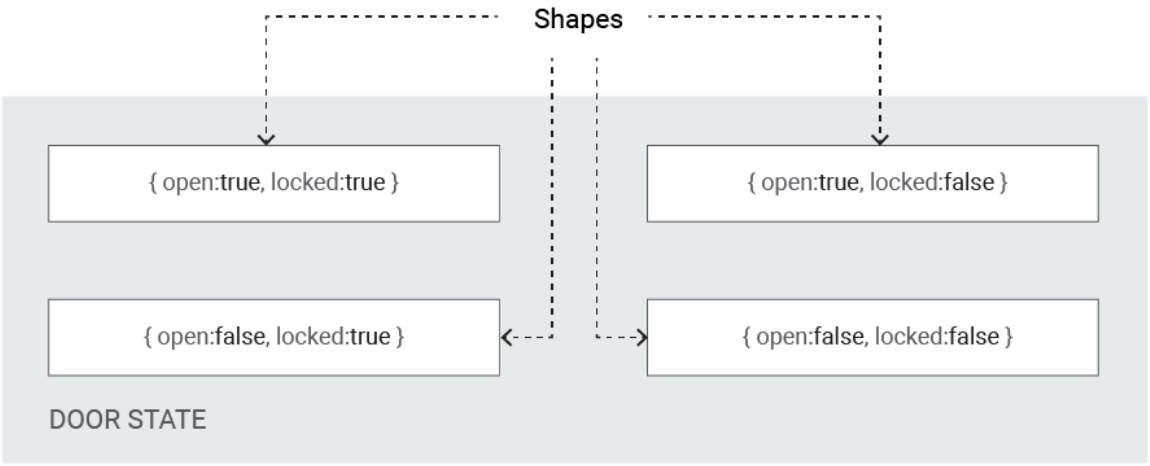

Figura 1.1: Insieme delle shapes del tipo DoorState

In Ballerina un tipo è, quindi, rappresentato dall'insieme delle possibili shapes che esso può avere.

Il concetto di shape permette di definire anche il concetto di **sottotipizzazione**. La sottotipizzazione in Ballerina è **semantica**: un tipo S è un sottotipo di T se l'insieme delle shapes denotate da S è un **sottoinsieme** delle shapes denotate da T. Considerando l'esempio precedente, è possibile creare, ad esempio, un nuovo tipo *EmergencyDoorState*, dove il campo *locked*, del record deve essere sempre impostato a falso.

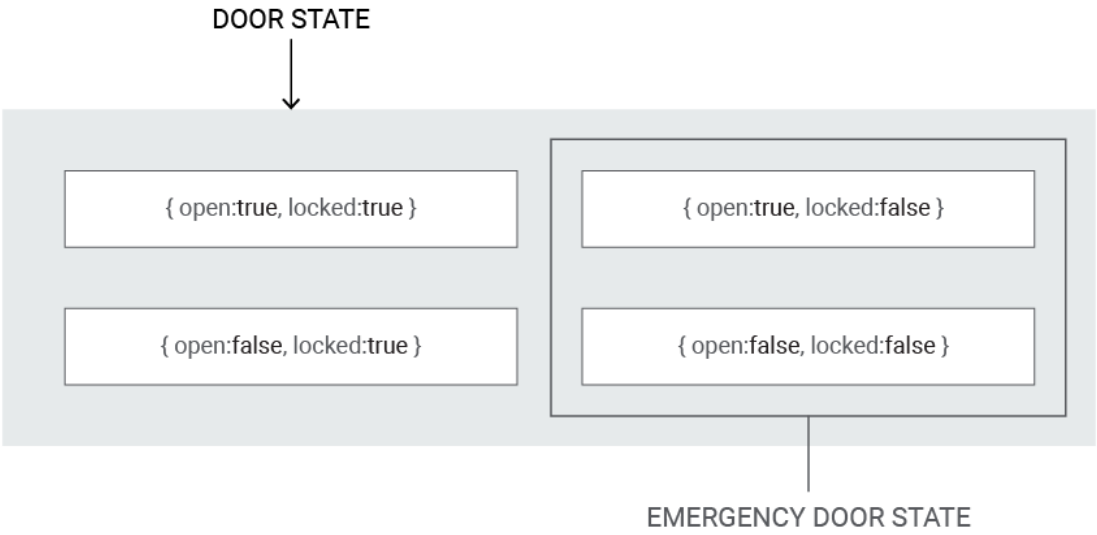

Figura 1.2: Insieme delle shapes del tipo DoorState e del tipo EmergencyDoorState

In questo caso il tipo EmergencyDoorState è denotato soltanto da un sottoinsieme delle shapes del tipo DoorState, quindi esso è un **sottotipo** del tipo DoorState. Dal momento che le shapes astraggono dall'identità di archiviazione dei valori, esse rappresentano degli alberi piuttosto che dei grafi.

1.4.4 Dati semplici

In ballerina un valore è considerato un **dato semplice** (plain data) se esso è un valore semplice, un valore sequenziale o strutturato che non contiene un valore comportamentale, ad ogni livello di profondità. In generale, un valore è considerato dato semplice nei seguenti casi:

- È un valore semplice,
- È un valore sequenziale,
- È un valore strutturato e tutti i suoi membri sono anch'essi dati semplici.

I Dati semplici possono essere confrontati per determinarne la loro uguaglianza.

1.4.5 Mutabilità

Esistono due tipi di elementi che possono mutare in Ballerina:

- I valori
- Le variabili.

La mutazione dei valori è legata all'*identità di archiviazione*: la mutazione può avvenire solo per quei tipi di valori che hanno un'identità di archiviazione. Quando un valore memorizzato in una certa locazione di memoria cambia, il cambiamento sarà visibile a tutte le variabili che fanno riferimento al valore in quella locazione di memoria. Non tutti i valori con un'identità di archiviazione, però, possono essere mutati: un valore potrebbe non consentire la mutazione, nonostante possegga un'identità di archiviazione.

La possibilità di mutazione dà luogo a due relazioni tra valori e tipi:

- Un valore **appare** come un tipo, in un particolare punto dell'esecuzione di un programma se la sua shape in quel momento è un membro di quel tipo.
- Un valore **appartiene** ad un tipo se esso *appare* come quel tipo e, necessariamente, continuerà ad apparire come esso, indipendentemente da come viene mutato il valore.

Se un valore non può essere mutato, allora le due relazioni sono equivalenti.

Quando un programma Ballerina dichiara una variabile in modo che abbia un tipo in fase di compilazione, questo significa che il compilatore e il sistema di runtime, assicurano che quella variabile potrà contenere soltanto un valore che appartiene a quel tipo.

Ogni valore ha quindi un **bit di sola lettura**. Se questo bit è impostato ad uno, allora quel valore è **immutabile**. Il bit di sola lettura viene fissato al momento della costruzione di ogni valore e non può essere cambiato successivamente.

Alcuni tipi elementari sono intrinsecamente immutabili: il bit di sola lettura è sempre attivo per un valore che appartiene ad un tipo elementare intrinsecamente immutabile. Tutti i valori semplici e le funzioni sono, ad esempio, intrinsecamente immutabili.

Altri tipi elementari che sono **selettivamente** immutabili: un tipo si definisce selettivamente immutabile se è possibile costruire sia valori mutabili che valori immutabili di quel tipo. Tutti i valori strutturati e gli oggetti sono selettivamente immutabili.

Infine alcuni tipi elementari sono intrinsecamente mutabili: il loro bit di sola lettura non può mai essere attivo per dei valori che appartengono ad un tipo intrinsecamente mutabile.

È, inoltre, possibile limitare la mutabilità di una variabile, rendendola finale. Questo comporta che una variabile dichiarata *final* non potrà più essere assegnata una volta inizializzata. In questo caso la variabile potrà comunque contenere un valore mutabile.

1.4.6 Isolamento

Esistono tre possibili operazioni sulla memoria:

- Lettura,
- Scrittura,
- Esecuzione.

Il concetto di immutabilità fa riferimento alla lettura e alla scrittura e fornisce informazioni limitate riguardo l'esecuzione: l'esecuzione non può portare alla mutazione di un valore immutabile. Per le funzioni e gli oggetti è utile avere più informazioni sul modo in cui le esecuzioni possano portare alla mutazione. A questo scopo Ballerina introduce il concetto di **isolamento**.

Ballerina definisce quando un valore è raggiungibile da una variabile: un valore è raggiungibile da una variabile se esso è raggiungibile dal valore contenuto nella variabile.

Gli oggetti, i metodi e le funzioni hanno un **bit di isolamento** in aggiunta al read-only bit. Un valore è **isolato** se il suo bit di isolamento è impostato ad uno. Ballerina garantisce che ogni stato mutevole, liberamente raggiungibile da un oggetto isolato o da una variabile isolata, venga acceduto solo tramite un istruzione di *lock*. Questo assicura che non ci saranno corse critiche accedendo ad un qualsiasi stato mutevole, che è raggiungibile tramite un oggetto isolato o una variabile isolata. Il funzionamento delle funzioni isolate verrà approfondito nella sezione 3.4.3

1.4.7 Iterabilità

Alcuni tipi elementari sono **iterabili**. Un valore iterabile è composto da zero o più valori più semplici e supporta un'operazione di **iterazione**, che fornisce i valori in sequenza uno dopo l'altro. L'insieme di valori che l'operazione di iterazione fornisce su un valore è la **sequenza di iterazione** di quel valore. Ogni tipo elementare definisce la sequenza di iterazione per i valori di quel tipo.

Esiste, inoltre, un valore associato al completamento dell'operazione di iterazione, chiamato **valore di completamento**, che può essere:

- **Nil** se l'iterazione viene completata con successo,
- **Un errore** se l'iterazione non viene completata con successo.

Ad ogni tipo di valore iterabile vengono, quindi, associati due tipi:

- Il tipo del valore, che rappresenta il tipo dei valori della sequenza di iterazione,
- Il tipo di completamento, che è il tipo del valore di completamento dell'iterazione.

La seguente tabella riassume i tipi elementari iterabili.

Tipi elementari	**sequenza di iterazione**	**Descrittore di tipo**	**Tipo del valore**	**Tipo di completamento**
string	sottostringhe di lunghezza 1	string	string:Char	()
xml	valori xml	xml<T>	T	()
list	membri della lista in ordine	T[]	T	()
mapping	membri	map<T>	T	()
table	membri in ordine	table<T>	T	()
stream	elementi dello stream	stream<T,C>	T	C

Tabella 1.1: Tipi elementari iterabili.

1.4.8 Valori sequenziali

I valori sequenziali appartengono ad uno dei due seguenti tipi elementari:

- **string**,
- **xml**.

Un valore sequenziale consiste in una **sequenza ordinata** di zero o più elementi costitutivi, i quali appartengono allo stesso tipo elementare del valore sequenziale stesso.

La **lunghezza** di un valore sequenziale è il numero di elementi costituenti. Ogni elemento costituente di una sequenza lunga n, ha un indice intero i, tale che $0 \geq i \geq n$. Per ogni tipo elementare sequenziale esiste un **valore vuoto**, che ha lunghezza zero e il cui tipo varia a seconda del tipo del valore sequenziale a cui appartiene. Il valore vuoto per il tipo string è quindi distinto dal valore vuoto del tipo xml, ed entrambi sono distinti da *nil*.

I valori appartenenti a un tipo elementare sequenziale X, possono essere definiti in termini di valori **singleton** e un operazione di concatenazione, come segue: (Con singleton si fa riferimento ad un insieme con un solo elemento).

- Il valore singleton di X appartiene ad X
- Il valore vuoto di X appartiene ad X
- Se x1, x2 appartengono a X, allora la concatenazione di x1 e x2 appartiene ad X

La concatenazione di qualsiasi valore $x \in X$ con la sequenza vuota di X, corrisponde ad **x**. Infatti una sequenza composta da un solo elemento x (singleton), è esattamente equivalente ad x.

I tipi degli elementi costituenti di un tipo elementare sequenziale di tipo X, sono quindi dei **sottotipi** di X. Questa è una differenza fondamentale tra liste e sequenze.

Solo i valori singleton di un tipo sequenziale possono avere un'*identità di archiviazione*. Quando un elemento costituente di un valore sequenziale ha un'identità di archiviazione, quello che viene memorizzato nella sequenza è un riferimento alla locazione di memoria dove il valore è memorizzato.

Un valore sequenziale è iterabile: la sequenza di iterazione consiste nei valori singleton ordinati della sequenza e il valore di completamento è sempre *nil*.

1.4.9 Unioni

Ballerina permette di definire dei tipi personalizzati basati sulla combinazione di due o più tipi. Un'**unione** di due tipi T1 e T2 è, quindi, l'unione dell'insieme delle shapes indicate da T1 e l'insieme delle shapes indicate da T2. Un valore X appartiene ad un unione di T1 e T2 se e solo se la sua shape è contenuta in T1 o in T2. Un'unione tra più tipi può essere definita utilizzando la notazione"T1|T2". Allo stesso modo è possibile definire una variabile di tipo *T?* che può contenere valori di tipo T oppure valori di tipo *Nil*.

```
type flexType string|int;

flexType a = 1;

flexType b = "Hello";
```

In questo esempio viene definita una variabile *flexType*, che può contenere valori di tipo *string* o *int*.

È possibile applicare il concetto di unioni anche ai dati strutturati. Durante l'esecuzione è poi possibile controllare l'attuale tipo del valore contenuto all'interno di una variabile, utilizzando l'operatore "**is**".

```
if x is string {
        return x.toUpperAscii();
    } else {
        return x;
    }
```

In questo esempio viene utilizzato l'operatore *is*, per controllare il tipo del valore contenuto all'interno della variabile.

1.4.10 Intersezioni

Allo stesso modo delle *unioni*, Ballerina permette di definire delle **intersezioni**. L'intersezione di due tipi T1 e T2 è l'intersezione delle shapes indicate da T1 e delle shapes indicate da T2. Un valore X, quindi, appartiene ad un intersezione di T1 e T2 se e solo se la sua shapes è contenuta sia in T1 che in T2.

Le intersezioni sono particolarmente utili per l'utilizzo del tipo **readonly**. Un'intersezione di più tipi può essere definita utilizzando la notazione "T1 & T2".

```
type TimeZone readonly & object {
    function getOffset(decimal utc) returns decimal;
};
```

1.4.11 Valori strutturati

I valori strutturati sono **contenitori** di altri valori, chiamati **membri**. Esistono tre tipi elementari di valori strutturati:

- Liste (List),
- Mappe (Mapping),
- Tabelle (Table).

I valori strutturati, normalmente, sono **mutabili**. Mutare un valore strutturato comporta il cambiamento dei valori che esso contiene. È comunque possibile creare valori strutturati immutabili e in questo caso l'immutabilità è profonda: un valore strutturato immutabile non può contenere valori strutturati mutabili.

Le **shapes** dei membri di un valore strutturato contribuiscono a formare la shape del valore strutturato stesso. Il cambiamento di un membro di un valore strutturato può, quindi, causare il cambiamento della sua shape.

Un valore strutturato ha un suo **tipo intrinseco**, che è un descrittore di tipo che fa parte del valore strutturato in fase di esecuzione. Il valore strutturato impedisce qualsiasi mutazione che potrebbe portare il valore strutturato ad avere una shape che non è un membro del suo tipo intrinseco. Quindi, un valore strutturato appartiene a un tipo se e solo se il suo tipo intrinseco è un **sottotipo** di quel tipo.

Ogni valore strutturato ha una **lunghezza**, che corrisponde al numero dei suoi membri. Tutti i valori strutturati sono **iterabili**: la sequenza di iterazione consiste nei membri della struttura e il valore di completamento è sempre *nil*.

I valori strutturati consentono accesso diretto ai loro membri tramite una **chiave**, che identifica univocamente i membri all'interno della struttura. Una chiave può essere:

- Out-of-line, se la chiave è indipendente dal suo membro;
- In-line, se la chiave è parte del suo membro.

1.5 Valori comportamentali

1.5.1 Funzioni

Una **funzione** è una parte di un programma che può essere eseguita esplicitamente. In ballerina, una funzione è anche un **valore**. Questo implica che una funzione può essere memorizzata in una variabile ed essere utilizzata come parametro o valore di ritorno di altre funzioni. Quando una funzione viene eseguita, prende in input una lista di argomenti (*param-list*) e restituisce in output un valore.

La *lista di argomenti* che viene passata ad una funzione può essere composta da zero o più argomenti in ordine. Ogni argomento è un valore ma la lista di argomenti non viene passata come valore. È compito del compilatore effettuare il **controllo del tipo** dei valori per accertarsi che la lista di argomenti passata alla funzione sia conforme alla param-list.

Un **defaultable-param** è un parametro per il quale viene specificato un valore di default. È possibile utilizzare un'espressione per specificare il valore di default e questa può anche effettuare dei riferimenti ai parametri precedenti. La caratteristica di un parametro di essere *defaultable* e il suo valore di default, non influiscono nella shape della funzione e nel suo tipo.

Quando l'esecuzione di una funzione ritorna al suo chiamante, essa ritorna esattamente un valore. Una funzione che non ritorna nessun valore viene rappresentata, in Ballerina, tramite una funzione che ritorna () (*nil*).

1.5.2 Oggetti

Un valore di tipo **oggetto**, incapsula dati e funzioni che operano con dei dati. Un oggetto consiste in un insieme di membri nominativi, dove ogni membro può essere un **campo** o una **funzione**. Un campo di un oggetto memorizza un valore. I metodi e i campi di un oggetto hanno dei nomi che li identificano univocamente all'interno dell'oggetto.

Metodi

Un **metodo** di un oggetto è una funzione che può essere invocata sull'oggetto tramite una *chiamata a metodo*. Quando un metodo viene invocato, esso può accedere all'oggetto tramite la variabile *self*.

I metodi di un oggetto vengono associati ad esso quando l'oggetto viene costruito e non possono più essere modificati.

Interazione di rete

Gli oggetti sono la base per le primitive di **interazione di rete** di Ballerina. Esistono due specifici tipi di oggetti per le interazioni di rete:

- Oggetti **client**,
- Oggetti **service**.

Un oggetto di tipo *client*, supporta l'interazione di rete con un sistema remoto originato dal **programma Ballerina**.
Un oggetto di tipo *service*, supporta l'interazione di rete con un sistema remoto originato dal **sistema remoto** stesso.
Un oggetto può essere di tipo client, service o nessuno dei due. Non può però essere sia di tipo client che di tipo service. Un oggetto di tipo client o service viene definito come **oggetto di interazione di rete**. Questi speciali tipi di oggetto, forniscono due funzionalità addizionali:

- **Metodi remoti**,
- **Risorse**.

Un *metodo remoto*, è un metodo utilizzato per le interazioni di rete e si differenzia in base al tipo di oggetto:

- Oggetti service: i metodi remoti vengono utilizzati per l'interazione di rete in **entrata**;
- Oggetti client: i metodi remoti vengono utilizzati per l'interazione di rete in **uscita**.

Le *risorse* sono, invece, utilizzate soltanto dagli oggetti service. Gli oggetti client, service ed il loro utilizzo verranno approfonditi nel capitolo 2

Shape

Una **shape** di un oggetto è composta da:

- Il bit di sola lettura,
- Il bit di isolamento,
- La tipologia di interazione di rete (client, service o empty),
- Un insieme non ordinato di shapes di campi,

- Un insieme non ordinato di shapes di metodi.

Le shape dei campi e dei metodi sono delle quadruple contenenti:

- Il **nome** del metodo o campo,
- La **regione di visibilità**,
- Un flag per indicare se si tratta di un metodo o campo **remoto**,
- La **shape** del valore del campo o della funzione del metodo.

Classi

È possibile descrivere un oggetto tramite una **classe**. Una classe descrive il tipo di un oggetto e fornisce un modo per costruire oggetti che appartengono a quel tipo. In particolare, essa fornisce le definizioni dei metodi associati all'oggetto quando esso viene costruito. Il tipo oggetto è selettivamente mutabile: è possibile specificare se un oggetto è di sola lettura utilizzando il descrittore di tipo *readonly*. Se un oggetto viene definito come di sola lettura, allora tutti i suoi campi dovranno essere di sola lettura e i loro valori non potranno essere cambiati dopo la creazione dell'oggetto.

```
class A {

    private int n;

    // Costruttore della classe
    public function init(int n = 0) {
        self.n = n;
    }

    public function inc() {
        self.n += 1;
    }

    public function get() returns int {
        return self.n;
    }
}
```

Gli oggetti possono essere, quindi, **inizializzati** in base ad una classe definita all'interno di un modulo. È possibile accedere ai campi di un oggetto oppure effettuare una chiamata ad un metodo di un oggetto, utilizzando la notazione "*oggetto.campo/funzione*".

```
function main( ) {
    m:A x = new m:A(1234);
    x.inc();
    int n = x.get();
}
```

Tipi oggetto

Ballerina permette, inoltre, di definire degli elementi di **tipo oggetto** senza specificare l'implementazione. A differenza delle classi, che forniscono sia il tipo che l'implementazione di un oggetto e attraverso le quali è possibile costruire degli oggetti, i *tipi oggetto* non forniscono l'implementazione. È possibile pensare ai tipi oggetto, in Ballerina, come a delle interfacce in altri linguaggi di programmazione.

```
type Hashable object {
    function hash() returns int;
};

function h() returns Hashable {
    var obj = object {
        function hash() returns int {
            return 42;
        }
    };
    return obj;
}
```

In questo esempio viene effettuato il controllo sull'oggetto ritornato (obj), per verificare che esso soddisfi la struttura dell'oggetto *Hashable*, il quale deve contenere una funzione chiamata *hash()*, che ritorna un valore intero. Se l'oggetto A soddisfa la struttura richiesta dall'oggetto B, allora esso può essere considerato come tipo B.

Sottotipizzazione

Come visto in precedenza, in Ballerina la sottotipizzazione è **semantica**: un oggetto T' è un **sottotipo** di un oggetto di tipo T se e solo se per ogni campo o metodo $f \in T$ esiste un corrispondente $f' \in T'$, tale che:

- Il tipo di f' è un **sottotipo** di f,
- La regione di visibilità di f' è uguale alla regione di visibilità di f.

È possibile anche includere dei *tipo oggetto* usando la sintassi "*T". Esistono due opzioni di utilizzo:

- Un tipo oggetto che include un altro tipo oggetto. In questo caso è possibile pensare ad un'interfaccia che **estende** un'altra interfaccia.
- Una classe che include un altro tipo oggetto. In questo caso è possibile pensare ad una classe che **implementa** un'interfaccia.

```
type A object {
    function f1() returns string;
};

type B object {
    *A;
    function f2();
};

class C {
    *B;

    function f1() returns string {
        return "C";
    }

    function f2() {
    }
}
```

Nell'esempio precedente A è un tipo oggetto (interfaccia), B è un altro tipo oggetto che include il tipo A (un'interfaccia che estende un'altra interfaccia) e C una classe che include B ed implementa i metodi definiti in A ed in B (una classe che implementa un'interfaccia).

Il controllo dell'implementazione da parte della classe del tipo oggetto che include viene effettuata in fase di compilazione. Questo meccanismo consente quindi l'ereditarietà delle interfacce. Ballerina non supporta l'ereditarietà delle implementazioni.

1.6 Gestione degli errori

In ballerina un valore di tipo errore fornisce **informazioni** riguardo l'errore che si è verificato. I valori di tipo errore appartengono ad un tipo elementare separato. Questo permette al linguaggio di gestire in maniera diversa gli errori dagli altri tipi di valori.

Un valore di tipo errore è **immutabile** e contiene le seguenti informazioni:

- Un **messaggio**, che è una stringa contenente informazioni che descrivono l'errore, in un formato leggibile all'uomo.
- Una **causa**, che può essere *nil* o un qualsiasi altro valore di tipo errore, che rappresenta la causa che ha scaturito questo errore.
- Dei **dettagli**, rappresentati da una mappa immutabile contenente informazioni aggiuntive riguardo l'errore. La mappa deve essere un sottotipo di *map<Clonable>*.
- Uno **stack trace**, cioè uno snapshot immutabile dello stato dello stack di esecuzione.

La shape del messaggio, della causa e dei dettagli sono parte della shape dell'errore. Lo stack trace non fa parte della shape dell'errore.

Ballerina non supporta la notazione delle eccezioni. Infatti, gli errori vengono gestiti come una normale parte del controllo di flusso. Esistono dei meccanismi appositi per gestire i valori di tipo errore. Il tipo *error* è un tipo elementare ed è quindi possibile utilizzarlo come un qualsiasi valore di un altro tipo. Ad esempio, le funzioni tengono traccia degli errori ritornando un valore di tipo errore.

```
function divide(int a, int b) returns int|error {
    if b == 0 {
        return error("Impossibile dividere per 0");
    }
    return a / b;
}
```

Nell'esempio precedente la funzione può ritornare sia un valore *error*, che un valore *int*. È possibile definire una funzione che ritorni esplicitamente soltanto valori error, usando la notazione *error?*. L'assenza di un errore deve infatti essere definita esplicitamente.

Quando un errore viene generato da una funzione, esso viene passato al suo chiamante. È possibile ussare l'operatore *is*, per controllore il tipo dell'errore e prendere delle decisioni in base ad esso. Alternativamente, l'errore viene passato al chiamante fino ad arrivare alla funzione *main*.

```
function intFromBytes(byte[] bytes) returns int|error {

    string|error ret = string:fromBytes(bytes);

    if ret is error {
        return ret;
    } else {
        return int:fromString(ret);
    }
}
```

Ballerina mette a disposizione, inoltre, la parola chiave **check**, in sostituzione dell'operatore *is* per utilizzare una sintassi più concisa.

```
function intFromBytes(byte[] bytes) returns int|error {

    string str = check string:fromBytes(bytes);

    return int:fromString(str);
}
```

In questo caso l'espressione seguita dalla parola chiave *check*, viene valutata (string:fromBytes(bytes)) e se la funzione *string:fromBytes()*, ritorna un errore allora la funzione *intFromBytes* ritorna immediatamente l'errore.

In ballerina esiste una distinzione tra errori **normali** e **anormali**. Quelli normali sono errori di cui, tipicamente, si ha il controllo e vengono gestiti

nella logica dell'applicazione. Alcuni errori, però, non sono sotto il controllo del programmatore e vengono chiamati errori anormali. In questa categoria rientrano, ad esempio, bug in codici di librerie oppure errori di *out of memory*. Questi tipi di errori non dovrebbero essere restituiti dalle funzioni e, infatti, non vengono esposti al programmatore. A differenza dei normali errori, che vengono gestiti tramite l'utilizzo del tipo *error*, gli errori anormali vengono gestiti tramite l'utilizzo dell'istruzione *panic*.

```
function divide(int a, int b) returns int {
    if b == 0 {
        panic error("Impossibile dividere per 0");
    }
    return a / b;
}
```

In questo caso l'istruzione *panic* comporta la terminazione immediata del programma. In generale, quando viene eseguita un operazione *panic*, la funzione attualmente in esecuzione viene fermata e terminata. Di seguito il flusso di esecuzione torna al chiamante di quella funzione. In questo modo, si continua seguendo lo stack di chiamate di funzione finché qualcuno non **intercetta** l'errore. Per intercettamento degli errori s'intende la necessità di rilevare un errore che è stato generato tramite un operazione di *panic*. È possibile specificare esplicitamente al programma di voler intercettare un errore tramite la sintassi:

```
error|T result = trap <expression>;
```

Nel caso in cui non si intercetti un errore di tipo *panic*, allora esso verrà propagato su tutto lo stack delle chiamate di funzione fino a terminare tutti i processi incontrati durante il suo cammino.

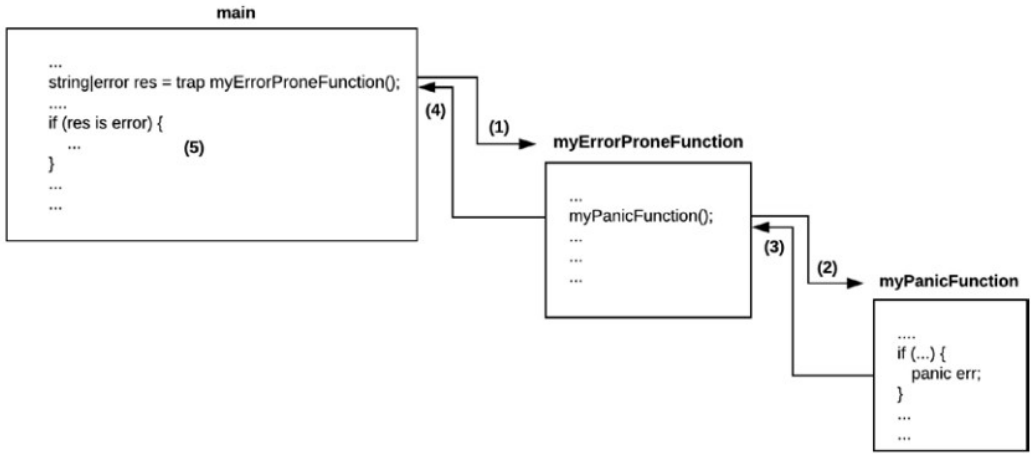

Figura 1.3: Flusso di gestione di un errore *panic* con *trap* [16]

In questa figura è possibile osservare come il flusso dell'esecuzione inizi nella funzione main e successivamente arrivi nella funzione *myPanicFunction*, dove avviene una situazione di *panic*. Questo causa, a sua volta, l'uscita immediata di tutte le funzioni chiamate , finché non si incontra un'istruzione *trap*. In questo caso l'errore di panic viene propagato all'indietro fino ad arrivare alla funzione main, che **intercetta** il panic con l'istruzione trap.

Capitolo 2

Cloud native

Ballerina mette a disposizione dei costrutti che si adattano perfettamente ai concetti di programmazione di rete, come **servizi** e **risorse**. Viene inoltre fornito, dal linguaggio, un supporto per distribuire le applicazioni su cloud, utilizzando Docker [17] o Kubernetes. In un'architettura a **microservizi**, vengono sviluppati, scalati e messi in esecuzione, individualmente, dei servizi di piccole dimensioni. Questi servizi disaggregati comunicano tra di loro tramite la **rete**, forzando gli sviluppatori a gestire gli errori e le problematiche dei sistemi distribuiti nella logica dell'applicazione. Le prossime sezioni illustreranno alcune capacità intrinseche di Ballerina per lo sviluppo di servizi distribuiti in modo efficace e il suo processo di sviluppo *cloud-native*, che viene fornito come parte dell'esperienza di programmazione.

2.1 Risorse

Similmente ai *metodi remoti*, negli oggetti di tipo *service* esiste un altro tipo di concetto, le **risorse**. Le risorse espongono i servizi in maniera *RESTful* [18], mentre i metodi remoti espongono i servizi in in maniera **procedurale**. In aggiunta, le risorse rappresentano delle entità a cui è possibile far riferimento tramite nomi, mentre è possibile far riferimento ai metodi remoti tramite verbi o azioni.

Le *risorse* consentono un approccio *data-oriented* per esporre l'interfaccia di un servizio. Esse sono abbastanza generiche da funzionare non solo con HTTP [19], ma anche con formati più moderni come GraphQL [20].

In ballerina è possibile definire dei metodi di tipo **resource** sia negli oggetti di tipo client che di tipo service.

```
import ballerina/http;

service /example on new http:Listener(8080) {

    resource function get hello(string name) returns string {
        return "Hello, " + name;
    }
}
```

In questo esempio viene creato un metodo *resource*. Invece di utilizzare la parola chiave *remote*, utilizza *resource*. La definizione è divisa in due parti:

- get: Viene utilizzato per accedere alla risorsa, che in questo caso rappresenta il metodo HTTP.
- name: Rappresenta il nome della risorsa.

Questo oggetto di tipo service definisce una risorsa "*name*", con un **parametro di query** chiamato "*param*" Questa definizione è simile ai *getter* ed i *setter* nella programmazione ad oggetti, ma generalizzata per la programmazione orientata alla rete (Network-oriented Programming).

Le risorse sono **gerarchiche**. Ognuna di esse ha un **percorso**, che è composto da un **percorso base** e da molteplici **segmenti**. Nell'esempio precedente, l'oggetto di tipo service è stato associato ad un HTTP *listener* tramite il percorso base "/example". Il service, inoltre, definisce un metodo *resource* "person/name". Quindi il percorso reale della risorsa, che verrà esposto all'esterno dal service sarà la combinazione del percorso base e dei segmenti che compongono il percorso della risorsa (example/person/name). I percorsi delle risorse possono, inoltre, essere **parametrizzati**. È possibile assegnare dei percorsi in modo dinamico durante la fase di accesso alla risorsa utilizzando la notazione "[ ]".

È possibile accedere a una risorsa di un oggetto **client** solo utilizzando un'azione di **accesso alla risorsa**. Per accedere ad una risorsa di un oggetto **service** viene, invece, utilizzato un oggetto *listener* fornito da un modulo di libreria.

```
import ballerina/io;

var clientObj = client object {
    resource function get greeting/[string name]() returns string {
        return "Hello, " + name;
    }

    resource function post game/[string name]/
    [int players]() returns string {
        return name + ": " + players.toString();
    }
};

public function main() {
    string name = "Mark";
    string result = clientObj
->/greeting/[name];
    // stampa Hello, Mark
    io:println(result);

    [string, int] gameDetails = ["Chess", 2];
    result = clientObj
->/game/[...gameDetails].post;
    // Stampa Chess: 2
    io:println(result);
}
```

Nell'esempio precedente la risorsa "*name*", viene acceduta tramite la sintassi "->/", utilizzata per effettuare una chiamata ad un metodo remoto. L'azione di accesso alla risorsa è di default "get", se essa non viene specificata. In questo esempio viene anche utilizzata la notazione "[...espressione]", per indicare che si vogliono utilizzare gli elementi presenti all'interno di una struttura dati.

2.2 Utilizzare servizi

Le applicazioni client utilizzano i **servizi di rete**. Pertanto, Ballerina supporta la definizione di oggetti client per consentire a un programma di interagire con servizi di rete, utilizzando i *metodi remoti* [21].

```
import ballerina/email;

function mailDemo() returns error? {

    email:SmtpClient clientObj = check new ("smtp.example.com",
                                            "user@example.com",
                                            "Passwd123");
    check clientObj->sendMessage({
        to: "contact@ballerina.io",
        subject: "Ballerina",
        body: "Hello!"
    });
}
```

In questo esempio, la funzione *mailDemo()*, usa un *email client* per inviare un'email. L'oggetto client (clientObj), in questo caso, appartiene alla classe *SmtpClient*. I parametri della connessione vengono passati al costruttore dell'oggetto client. Il metodo remoto *sendMessage()* viene chiamato sull'oggetto client, utilizzando la sintassi "->". Questa notazione specifica che si tratta di una chiamata remota ad un servizio di rete.

È importante evidenziare come Ballerina sia implicitamente **non bloccante** nelle interazioni di rete. Nel momento in cui viene effettuata una chiamata ad un metodo remoto, internamente, viene effettuata un'esecuzione **asincrona**, utilizzando l'I/O non bloccante, in cui il thread di esecuzione corrente viene rilasciato allo scheduler del runtime di Ballerina, per essere utilizzato da altri *strands*. Al termine dell'operazione di I/O, l'esecuzione del programma riprende automaticamente dal punto in cui era stata sospesa. Questo modello offre un pattern di programmazione molto più conveniente rispetto alla gestione manuale dell'I/O non bloccante, fornendo al tempo stesso la massima efficienza delle prestazioni.

Le chiamate a metodi remoti di questo tipo hanno alcune restrizioni:

- Non sono consentite nelle espressioni profondamente annidate;
- Esiste uno spazio dei simboli specifico per questi tipi di metodi;
- I metodi remoti sono implicitamente pubblici.

Tipicamente non è necessario dover scrivere le proprie classi per gli oggetti client, in quanto essi vengono forniti con dei moduli di libreria.

2.3 Offrire servizi

Mettere a disposizione un servizio è un attività complessa che coinvolge tre elementi principali:

- Un **oggetto service**: Questi oggetti sono definiti dall'applicazione e possono contenere metodi remoti che sono accessibili tramite la rete. Essi vengono chiamati da un **remote system**. Allo stesso modo i metodi remoti di un oggetto client chiamano il *remote system*;

- Un **oggetto listener**: Esso fornisce un'interfaccia tra la rete ed i servizi. Verranno approfonditi nella sezione 2.3.1;

- I **moduli**: In Ballerina i moduli hanno un **ciclo di vita** e vengono inizializzati all'avvio del programma. In seguito all'inizializzazione, i *listeners* registrati con quel modulo, vengono avviati, e quando il programma termina, anche i listeners vengono terminati.

I **metodi remoti** o i **metodi resource**, degli oggetti service utilizzano i loro valori di ritorno per indicare al listener quali ulteriori gestioni del messaggio sono necessarie. Il ritorno di un errore viene utilizzato per indicare che il metodo ha riscontrato un errore durante la gestione del messaggio di rete. Quando viene ritornato un oggetto service, si specifica al listener di dover utilizzare quel service per gestire ulteriormente il messaggio. Quando non è necessaria nessuna gestione ulteriore del messaggio, allora il valore di ritorno è *nil*.

Il **valore di ritorno** può, inoltre, essere utilizzato per fornire una risposta al messaggio di rete. Questo sistema però comporta delle limitazioni, in quanto, il metodo non può controllare cosa succede nel caso in cui dovesse esserci un errore durante l'invio del messaggio di risposta. Un'altra limitazione è quella di non poter gestire degli scambi di messaggi complessi, nonostante il ritorno di risposte multiple ad una singola richiesta possa essere gestito tramite l'utilizzo di *stream*. In ballerina, un oggetto listener può ovviare a queste limitazioni, passando un oggetto client come parametro dei metodi remoti o dei metodi *resource*. In questo modo un oggetto service può effettuare le chiamate sui metodi remoti degli oggetti client, al fine di fornire delle risposte al client.

2.3.1 Listeners

Come detto in precedenza un **listener** fornisce un interfaccia tra la rete ed i servizi. Un listener riceve **messaggi** di input di rete da un processo remoto, su un certo protocollo, poi traduce i messaggi ricevuti con delle chiamate ai

metodi remoti degli oggetti *service*. Gli oggetti service vengono associati a dei *listeners*, che sono, a loro volta, registrati in un *modulo*.

```
import ballerina/http;

listener http:Listener list = new (8080);
```

Come in questo esempio, una dichiarazione di un listener è molto simile alla dichiarazione di una variabile. La dichiarazione registra il listener con il modulo in cui esso viene dichiarato. Se l'espressione *new*, ritorna un errore, allora l'inizializzazione del modulo fallisce. Ogni listener è descritto dal seguente descrittore di tipo:

```
object {
   public function attach(T svc, A attachPoint) returns error?;
   public function detach(T svc) returns error?;
   public function start() returns error?;
   public function gracefulStop() returns error?;
   public function immediateStop() returns error?;
}
```

In questo caso **T** è un sottotipo di *service object* ed A è un sottotipo di *string[]|string|()*. Di seguito verranno illustrate le funzioni dei metodi di un listener:

- **attach**: viene utilizzato per fornire al listener un oggetto service, che potrà in seguito utilizzare per gestire i messaggi in arrivo sulla rete. L'argomento *attachPoint* specifica su quali messaggi di rete l'oggetto service deve essere utilizzato.

- **detach**: viene utilizzato per indicare al listener di smettere di utilizzare un oggetto service fornito in precedenza con una *attach*.

- **start**: viene utilizzato per comunicare al service di iniziare a gestire i messaggi di rete in entrata.

- **gracefulStop**: viene utilizzato per avvisare il listener di smettere di gestire delle nuove richieste in arrivo. In questo caso il listener deve completare la gestione di tutti i messaggi già presi in carico. In particolare deve aspettare il completamento di tutti gli *strand* creati per eseguire i metodi degli oggetti service.

- **immediateStop**: anche questo metodo viene utilizzato per avvisare il listener di terminare di gestire nuove richieste. A differenza di *gracefulStop*, però, il listener non deve tentare di completare le richieste già prese in carico. In particolare deve sopprimere ogni strand creato per eseguire i metodi dei service.

Ogni metodo ritorna un valore di errore nel caso in cui dovesse verificarsi un errore, altrimenti ritorna *nil*.

2.3.2 Ciclo di vita dei moduli

Come annunciato precedentemente, i **moduli** hanno un ciclo di vita. Durante l'inizializzazione del programma vengono registrati tutti i moduli listeners. La sequenza di inizializzazione avviene in un ordine tale che: se un modulo A importa il modulo B, allora il modulo A viene inizializzato **dopo** il modulo B. La fase di inizializzazione termina tramite la chiamata della funzione *main*, se presente.

Se esistono dei listeners registrati in un modulo, allora la fase di inizializzazione del modulo viene seguita dalla fase di **ascolto**. Questa fase inizia tramite la chiamata del metodo *start* su ogni listener registrato. Essa termina quando il programma viene terminato con un segnale (SIGINT, SIGTERM). Questo comporta, quindi, la chiamata delle funzioni *gracefulStop* o *immediateStop* su ciascun listener registrato. Il tipo di funzione scelto dipenderà dal tipo di segnale utilizzato per terminare il programma. Queste attività legate al ciclo di vita dei moduli, sono integrate all'interno di Ballerina.

I moduli, allo stesso modo degli oggetti, posseggono una funzione **init**. Questa funzione viene chiamata dopo che tutte le variabili all'interno di quel modulo vengono inizializzate.

```
function init() {
    io:println("X");
}
```

Il tipo di ritorno delle funzioni *init* deve essere un sottotipo di *error?*. Se un modulo ha una funzione main, è possibile assumere che la funzione *init()*, venga chiamata automaticamente prima della funzione *main*.

2.3.3 Dichiarazione

La possibilità di creare oggetti senza l'utilizzo di una classe è utile alla dichiarazione di oggetti service.

```
import ballerina/io;
import ballerina/udp;

listener udp:Listener u = new (8080);

udp:Service obj = service object {
    remote function onDatagram(udp:Datagram & readonly dg) {
        io:println("bytes received: ", dg.data.length());
    }
};

function init() returns error? {
    check u.attach(obj);
}
```

Nell'esempio, il corpo della dichiarazione dell'oggetto service definisce il *metodo remoto* "onDatagram". Nel metodo *init*, viene poi, esplicitamente, associato il service di tipo udp, al listener di tipo udp. Esiste anche un modo più conciso per dichiarare oggetti di tipo service:

```
import ballerina/io;
import ballerina/udp;

service on new udp:Listener(8080) {

    remote function onDatagram(udp:Datagram & readonly dg) {
        io:println("bytes received: ", dg.data.length());
    }
}
```

In questo caso, il **tipo** dell'oggetto service viene determinato in base al tipo del listener. L'oggetto di tipo *udp:listener*, accetta infatti dei service di tipo *udp:Service*, con un metodo remoto *onDatagram* che accetta dei parametri di tipo *udp:Datagram & readonly*.

2.4 Query native

Ballerina fornisce il supporto per effettuare la **list comprehension**, tramite una sintassi simile a quella di SQL [22]. Per *list comprehension* s'intende la creazione di liste a partire da altre liste. Un'espressione di una query segue, in generale, la semantica *XQUERY FLWOR* [23] (for, let, where, order by, and return). È possibile immaginare la semantica di una query come una **pipeline**, che inizia generando un elenco di associazioni nella prima fase. Di seguito, le fasi successive prendono le associazioni dalla fase precedente della pipeline e generano in output un altro set di associazioni di variabili.

```
int[] nums = [1, 2, 3, 4];

int[] numsTimes10 = from var i in nums
    select i * 10;

int[] evenNums = from var i in nums
    where i % 2 == 0
    select i;
```

Nell'esempio, l'array *numTimes10*, è costruito iterando sull'array *num* e utilizzando la clausola *select* per moltiplicare i valori della lista per 10. Allo stesso modo è possibile applicare dei filtri in stile SQL utilizzando la clausola *where*. Nell'esempio, viene implementato un filtro per selezionare soltanto i numeri pari, utilizzando una clausola *where*. È, inoltre, possibile ordinare gli elementi della lista utilizzando la clausola *order by*. L'ordinamento funziona in modo consistente con gli operatori *<*, *<=*, *>*, *>=*. Esistono dei casi più complessi dove non è possibile ordinare i dati. Per Ballerina qualsiasi confronto con *nil*, o con un valore floating point di tipo *NaN*, è **disordinato**. I valori disordinati che vengono incontrati durante una query, vengono ritornati alla **fine** della lista.

2.5 Integrazione con Docker, Kubernetes

Kubernetes è una piattaforma utilizzata per eseguire applicazioni con diversi **microservizi**. Può essere utilizzato per la messa in produzione automatica, la scalabilità dei prodotti e per la gestione delle applicazioni containerizzate. Kubernetes definisce un insieme di blocchi univoci predefiniti, che devono essere definiti in un file YAML e distribuiti nel cluster Kubernetes. In alcuni casi, creare questo file YAML, può essere impegnativo e, per questo, il compilatore Ballerina può occuparsi della creazione di questo file, durante la compilazione

del codice sorgente. Il seguente esempio di codice mostra le opzioni necessarie durante la fase di **build** dell'applicazione, per consentire al compilatore di generare il file YAML che consente di eseguire il codice su Kubernetes.

```
import ballerina/http;

service /hello on new http:Listener(9090) {
    resource function get sayHello() returns string {
        return "Hello!";
    }
}
```

Eseguendo il *build* del file sorgente con il comando "*bal build –could=k8s*", il compilatore genererà il file YAML di kubernetes e l'immagine **Docker**, richiesti per la messa in esecuzione dell'applicazione all'interno di Kubernetes.

Capitolo 3

Concorrenza

Uno degli aspetti chiave del linguaggio Ballerina è quello della **concorrenza** [24]. Con l'aumentare delle applicazioni che richiedono l'interazione di rete, la concorrenza è diventata un aspetto fondamentale, ma allo stesso tempo, essa introduce una maggiore complessità nella gestione dei dati.

Una delle idee principali di Ballerina è quella di avere una rappresentazione grafica del programma. Questo concetto ha lo scopo di fornire informazioni più approfondite su ciò che il programma sta facendo. Un **diagramma di sequenza**, infatti, fornisce una visione concisa del programma, coinvolgendo sia l'interazione di rete, che la concorrenza. Questo aspetto del linguaggio verrà approfondito nella sezione 4.5. Le funzionalità relative alla concorrenza sono integrate all'interno del linguaggio e di seguito ne verranno illustrate alcune, tra quelle principali.

3.1 Workers

Il linguaggio permette di definire un **flusso di controllo concorrente**, utilizzando un **named worker**. Esso viene definito utilizzando la parola chiave *worker*. Normalmente il codice di una funzione appartiene al worker della funzione, che ha un singolo thread logico di controllo. È possibile, però, che una funzione dichiari un named worker, che viene eseguito *concorrentemente* al worker di default della funzione e agli altri workers.

Un named worker non può mai iniziare la sua esecuzione prima della sua dichiarazione. Questo comporta che il codice precedente alla dichiarazione del worker viene sempre eseguito **prima**, che il worker inizi ad essere eseguito. Le variabili dichiarate prima di tutti i workers e i parametri della funzione sono comunque accessibili all'interno dei workers.

```
import ballerina/io;

public function main() {

    io:println("Before workers A and B");

    worker A {
        io:println("In worker A");
    }

    worker B {
        io:println("In worker B");
    }

    io:println("In function worker");
}
```

Nell'esempio, la funzione *main()*, ha un worker di default e al suo interno vengono definiti due workers A e B. Il codice dei due workers all'interno della funzione, verranno eseguiti concorrentemente al codice del worker di default della funzione stessa.

I workers possono continuare ad essere eseguiti anche dopo che il worker di default termina l'esecuzione e dopo il ritorno della funzione. Se si vuole **aspettare** che un worker X termini, è possibile indicare esplicitamente al worker che si vuole far aspettare di attendere la terminazione del worker X utilizzando l'espressione *wait X*.

```
import ballerina/io;

public function main() {
    io:println("Before worker A");

    worker A {
        io:println("In worker A");
    }

    io:println("In default worker");
    wait A;
    io:println("After wait A");
}
```

In questo esempio, la parola chiave *wait*, viene utilizzata per indicare al worker di default della funzione *main()*, di aspettare la terminazione del worker A prima di effettuare il ritorno. È, inoltre, possibile effettuare la wait su più workers.

I workers possono avere un valore di ritorno proprio come una funzione. Questo valore di default è *nil*. Allo stesso modo delle funzioni, è possibile utilizzare la parola chiave *check* per gestire gli errori.

In ballerina il concetto di **future**, corrisponde al concetto di worker. Un worker definito come una variabile diventa un future. Il valore di ritorno del worker diventa il valore di ritorno del *future*.

```
function demo() returns future<int> {
    worker A returns int {
        return 2022;
    }
    return A;
}
```

In questo esempio, la funzione *demo()*, ritorna un future, che è il worker A e il tipo del future corrisponde al tipo di ritorno del worker, ovvero *int*.

Ballerina permette lo scambio di **messaggi** tra i workers utilizzando la notazione "->, <-".

```
public function main() {

    worker A {
        1 -> B;
        2 -> C;
    }
    worker B {
        int x1 = <- A;
        x1 -> function;
    }
    worker C {
        int x2 = <- A;
        x2 -> function;
    }

    int y1 = <- B;
    int y2 = <- C;
    int z = y1 + y2;
}
```

Nell'esempio di codice precedente, il worker A invia il valore intero 1 al lavoratore B e il valore intero 2 al lavoratore C, utilizzando la notazione "->". Sia B che C ricevono i valori e li memorizzano rispettivamente nelle variabili x1 e x2, usando la notazione "<-". Successivamente li inviano al worker di default della funzione principale, che li riceve tramite "<-" e calcola l'addizione dei due numeri interi. Tutti i messaggi vengono **copiati** utilizzando la funzione *clone()*, mentre i valori immutabili vengono passati senza effettuare una copia.

3.2 Strands

La gestione della concorrenza in Ballerina, basata sui workers, introduce un concetto noto come **strand**. Uno strand è un thread logico di controllo "leggero", assegnato ad ogni worker. L'esecuzione di uno strand può essere cambiata dallo scheduler, soltanto in alcuni casi specifici, come nelle *wait* o durante le chiamate bloccanti di sistema. Questo approccio evita la necessità di bloccare le variabili a cui si accede tra più workers, per gestire ed evitare problemi relativi a corse critiche o deadlock.

In ballerina, quindi, un *thread* può essere composto da più *strands* e all'interno di un singolo thread, può essere eseguito soltanto uno strand alla volta. Il **multitasking** sui vari strands di uno stesso thread, viene effettuato in maniera **cooperativa**. Nonostante uno strand abbia un thread logico di controllo, l'attuale esecuzione avviene comunque su un unico thread fisico a livello di CPU. Tuttavia, è possibile utilizzare un'annotazione per permettere allo strand di essere eseguito su un thread diverso.

3.3 Transazioni

Le **transazioni** sono un aspetto importante della concorrenza in Ballerina. Il sistema di runtime ha un supporto integrato per interagire con il **transaction manager**. Il sistema, infatti, include un transaction manager e il linguaggio fornisce la sintassi per definire le transazioni. La transazione corrente è parte del contesto di esecuzione di uno strand. Inoltre, il concetto di transazione può essere combinato con le funzionalità di interazione di rete, per fornire il supporto alle transazioni distribuite.

```
transaction {

    doStage1();
    doStage2();

    check commit;
}
```

Nell'esempio precedente, viene definito un blocco **transazione**, utilizzato per eseguire due chiamate a funzione che devono essere parte della transazione. La parola chiave **transaction** viene utilizzata per avviare la nuova transazione, mentre la parola chiave **commit**, deve essere inclusa esplicitamente per eseguire la transazione.

In ballerina, esistono quattro modi in cui è possibile uscire da un blocco *transazione*. Nei casi normali, il passaggio attraverso un'operazione di *commit* o di *rollback*, comporta nella terminazione della transazione. Gli altri due casi sono quelli in cui vengono incontrati degli errori e l'uscita può essere causata da un'operazione di *check* o da un errore di tipo *panic*.

Il linguaggio consente di porre un vincolo su delle funzioni, richiedendo che esse vengano chiamate soltanto in un **contesto transazionale**. Questo comporta, inoltre, che il corpo di queste funzioni sarà, esso stesso, un contesto transazionale. È possibile ottenere questo comportamento utilizzando il qualificatore **transactional**.

Il design di ballerina prevede che le transazioni lavorino con l'interazione di rete. Pertanto, i metodi remoti e i metodi resource di un oggetto service, possono essere dichiarati *transazionali*, ma l'effettivo funzionamento del comportamento transazionale dipende dall'implementazione e viene tenuto nascosto per evitare complicazioni.

Le transazioni seguono uno **schema di ramificazione** che parte da una transazione globale e prosegue, quindi, con più transazioni che si diramano da essa. Pertanto, la transazione corrente è sempre un **ramo della transazione globale**. Quando una nuova transazione viene creata come transazione globale, la transazione corrente diventa il ramo principale.

È possibile avere oggetti client e oggetti listener che sono "a conoscenza" delle transazioni. Per comunicare con i sistemi remoti in modo orientato alle transazioni, devono associare i messaggi di rete a una transazione globale e consentire al transaction manager del programma Ballerina di comunicare con altri transaction managers. Per questo, è necessario un **protocollo** per comunicare tra più programmi distribuiti. Questo non si limita a due programmi Ballerina, ma può funzionare con programmi scritti in linguaggi diversi o da Ballerina a un database, a condizione che entrambe le parti conoscano lo stesso

protocollo, inclusi i protocolli standard del settore come XA [25]. Ballerina ha un **protocollo di microtransazione** per supportare questa interazione ed è possibile implementarlo in altri linguaggi di programmazione.

3.4 Sicurezza nella concorrenza

3.4.1 Lock

L'istruzione **lock**, permette l'accesso di uno stato mutabile da diversi *strands*, eseguiti su thread separati, con dei risultati ben definiti.

```
int n = 0;

function inc() {
    lock {
        n += 1;
    }
}
```

Nell'esempio precedente il blocco incluso nel *lock*, garantisce la mutazione sicura della variabile n, da parte di diversi strands eseguiti su thread separati. La semantica del blocco di *lock*, può essere considerata come una sezione **atomica**.

3.4.2 Concorrenza nei servizi

L'obiettivo principale di Ballerina è quello di ottenere delle *performance discrete* ed un *discreto livello di sicurezza*. Con performance discrete s'intende che i listener possono avere diversi threads che **servono** le richieste in arrivo **concorrentemente**. Un discreto livello di sicurezza, similmente, può essere ridotto all'aspettativa che non ci siano corse critiche non rilevate, che portano a risultati errati. Tuttavia alcuni errori non individuati a compile-time, possono essere individuati a runtime. Risulta piuttosto difficile individuare tutti gli errori a compile-time, poiché questo richiederebbe un sistema molto più complesso e restrittivo. L'istruzione di *lock* non è sufficiente per ottenere il livello di sicurezza richiesto da Ballerina, dato che la scelta del suo utilizzo è lasciata allo sviluppatore. Pertanto, ci sono disposizioni aggiuntive in Ballerina che forniscono il livello di protezione che è in linea con la sicurezza richiesta.

3.4.3 Funzioni isolate

Per ottenere gli obiettivi di sicurezza richiesti, Ballerina offre il concetto di funzioni **isolate**. Una funzione isolata è una funzione che è **concurrency safe** se i suoi argomenti sono safe. Una funzione isolata può accedere ad uno stato mutevole solo attraverso i suoi parametri. Inoltre, può chiamare solo funzioni che sono a loro volta isolate.

```
type Entry map<json>;
type RoMap readonly & map<Entry>;

final RoMap map = loadMap();

function loadMap() returns RoMap {
    ...
}

isolated function lookup(string s) returns readonly & Entry? {
    return map[s];
}
```

In questo esempio, vengono definiti due tipi di mappe, *Entry* e *RoMap*. La mappa RoMap è anche sottotipo del tipo *readonly*. La variabile *map* è, invece, **finale**, indicando il fatto che essa non può essere riassegnata. Il valore contenuto all'interno della variabile *map* è anche di tipo readonly ed è, quindi, profondamente **immutabile**. Questo comporta che l'accesso alla variabile *map* da parte di più thread è completamente **sicuro**. Pertanto, quando si accede alla funzione isolata *lookup()* senza un'istruzione lock, essa restituisce un valore del tipo *Entry* che è anche di tipo readonly, garantendo così la completa sicurezza della concorrenza. In questa maniera, il tipo readonly integra le funzioni isolate.

Capitolo 4

Caso di studio

4.1 Introduzione

Nelle prossime sezioni verrà descritta l'implementazione di un caso di studio, che permetterà di evidenziare gli aspetti caratteristici del linguaggio e il modo in cui Ballerina renda semplice ed efficiente lo sviluppo di applicazioni cloud e di microservizi. È importante sottolineare, come Ballerina fornisca tutte le funzionalità necessarie allo sviluppo, al mantenimento, all'esecuzione e al monitoraggio di un servizio web, **integrate nel linguaggio**, senza il bisogno di dover utilizzare delle librerie esterne.

Il caso di studio in questione è un semplice **servizio REST** per la conversione di valute. Data una valuta iniziale, una valuta finale ed un valore, il servizio deve ritornare il valore convertito dalla valuta iniziale, nella valuta finale. A questo scopo è necessario che il servizio utilizzi, a sua volta, un altro servizio esterno, per recuperare i più recenti tassi di scambio delle valute interessate, necessari per effettuare la conversione. Vedremo, quindi, in che modo Ballerina consente di effettuare delle chiamate a dei servizi esterni e le funzionalità che fornisce, per rendere efficiente questo processo.

4.2 Configurazione

Ballerina permette di definire, a livello di modulo, delle variabili **configurabili**. Una variabile di questo tipo, consente di effettuare la sua inizializzazione a runtime, con dei valori iniziali specificati in un **file di configurazione**. È possibile assegnare un valore di default a una variabile configurabile, utile nel caso in cui non venga trovato nessun valore nel file di configurazione. Per quelle variabili, invece, per cui è obbligatorio specificare un valore iniziale nel

file di configurazione, è possibile utilizzare la sintassi "*?*", come inizializzatore della variabile.

In questo caso viene definita una **variabile configurabile** *port*, che rappresenta il numero della porta su cui il servizio sarà in ascolto una volta avviato e una variabile configurabile *name*, che rappresenta il nome del servizio.

```
import ballerina/http;
import ballerina/io;

configurable int port = 9090;
configurable string name = ?;
```

È necessario, innanzitutto, importare i **moduli necessari**. In questo caso sarà necessario il modulo *http* per la creazione del servizio e il modulo *io*, per effettuare delle stampe. Come detto in precedenza la variabile *port*, può anche non avere nessun valore specificato nel file di configurazione e in quel caso, verrà inizializzata con il valore specificato nel modulo (in questo caso 8080). Notiamo inoltre come la variabile *name*, invece, necessita obbligatoriamente la presenza di un valore iniziale nel file di configurazione.

A questo punto, per fornire dei valori iniziali, è necessario creare un file di configurazione (Config.toml) dove andranno definiti i valori iniziali delle variabili configurabili.

```
# File Config.toml
port = 8080

name = "RestApi"
```

Essendo presente un valore, anche per la variabile *port*, essa verrà inizializzata con il valore definito nel file di configurazione.

4.3 Open by default

Come detto in precedenza, per effettuare la conversione tra più valute, è necessario recuperare i tassi di scambio più recenti per le valute interessate. Per recuperare queste informazioni, nel momento in cui verrà effettuata una richiesta al servizio di conversione, sarà necessario effettuare una **chiamata** ad un servizio esterno per poter recuperare i tassi di scambio. È, quindi, utile poter memorizzare le informazioni ricevute dal servizio di API esterno in una struttura dati, che consentirà successivamente di accedere in modo semplice alle informazioni utili per la conversione. Per costruire il **record** che conterrà i dati relativi ai tassi di scambio richiesti, in modo coerente con le

informazioni ricevute dalla richiesta effettuata, è opportuno andare a studiare la documentazione del servizio di API esterno.

```
{
    "success": true,
    "timestamp": 1519296206,
    "base": "EUR",
    "date": "2022-11-11",
    "rates": {
        "AUD": 1.566015,
        "CAD": 1.560132,
        "CHF": 1.154727,
        "CNY": 7.827874,
        "GBP": 0.882047,
        "JPY": 132.360679,
        "USD": 1.23396,
    [...]
    }
}
```

È possibile osservare come il file json restituito dal servizio di API esterno, contenga più informazioni di quelle necessarie al nostro servizio per effettuare la conversione. Ballerina semplifica l'interazione di rete, adottando il principio **Open by default**. L'idea è quella di progettare programmi che interagiscono con la rete, che accettino tutti i dati che gli vengono inviati e che facciano il massimo sforzo per comprenderli. Il principale meccanismo che Ballerina fornisce, a questo scopo, è quello dei **record aperti**: Un record aperto è un tipo di dato che consente di aggiungere altri campi oltre a quelli specificati. Il tipo di questi campi non specificati è **anydata**. In questo caso, quindi, è stato creato un record *ConversionResponse*, definendo soltanto le informazioni strettamente necessarie al servizio. Successivamente a runtime, grazie al fatto che abbiamo definito un record aperto, nel momento in cui verranno ricevuti i dati dal servizio di API esterno e verranno convertiti nel tipo di dato *ConversionResponse*, saranno aggiunti al record dei campi non specificati, consentendo comunque la conversione nel tipo di dato desiderato, in maniera trasparente.

```
type ConversionResponse record {
   boolean success;
   map<decimal> rates;
};
```

In questo caso le uniche informazioni necessarie che andranno specificate nel record sono:

- **success**: un booleano che è false se è stato riscontrato un errore durante la nostra richiesta, true altrimenti.
- **rates**: una mappa contente il tasso di conversione per ogni valuta.

4.4 Creazione del servizio

Una volta definite le strutture dati e i valori di configurazione iniziali è possibile creare il service che andrà a gestire le richieste.

```
service / on new http:Listener(port) {

   resource function get convert/
   [string base]/to/[string target](decimal amount) returns decimal|error {
       http:Client exchangeAPI = check new ("https://api.exchangerate.host");
       ConversionResponse res = check exchangeAPI->get(string `/latest?base=${base}`);
       io:println(res);

       if !res.success {
           return error("Exchange rates couldn't be obtained");
       }

       decimal? rate = res.rates[target];
       if rate is () {
           return error("Couldn't determine exchange rate for target currency",
            targetCurrency = target);
       }
       return rate * amount;
   }
}
```

È stato, quindi, definito un oggetto di tipo **service** ed è stato associato ad un **listener** di tipo http. Questo listener è stato, poi, messo in ascolto sulla porta specificata dalla variabile *port*. È possibile osservare come Ballerina renda semplice e immediata la creazione di un oggetto di tipo service e la sua associazione ad un listener, che si occuperà di ricevere le richieste e di convertirle in chiamate ai metodi resource dell'oggetto service. In questo esempio, viene definito un unico metodo *resource*, che è quello che si occupa di ricevere le richieste e di restituire il valore convertito o eventuali segnalazioni di errori.

Come già detto in precedenza, nel momento in cui il servizio riceve una richiesta è necessario effettuare una richiesta ad un servizio di API esterno. Ballerina consente di effettuare questa operazione in poche righe di codice, grazie alla presenza degli **oggetti client**. In questo esempio, viene creato un oggetto client di tipo *http*, con riferimento al URL del servizio di API esterno

("https://api.exchangerate.host"). Gli oggetti client consentono, grazie ai metodi remoti, di effettuare delle chiamate a dei servizi di rete in modo efficace e **non bloccante**. Infatti, il thread di esecuzione corrente viene rilasciato allo scheduler di runtime di Ballerina per essere utilizzato da altri *strands*. Una volta ricevuta la risposta, l'esecuzione del programma riprende automaticamente dal punto in cui era stata sospesa. Il contenuto della risposta viene poi convertito in un record di tipo *ConversionResponse*, in modo che possa essere utilizzato, successivamente, per controllare lo stato della risposta e per effettuare la conversione della valuta, nel caso in cui non ci siano stati degli errori.

4.5 Rappresentazione grafica

Ballerina consente di generare, in modo automatico, una rappresentazione grafica di un programma, sotto forma di **diagramma di sequenza**. Questo diagramma mostra sia la logica, sia le interazioni di rete di una determinata funzione o risorsa di un servizio. Uno dei grandi vantaggi di avere una rappresentazione grafica di un programma è che essa agisce come documentazione del codice. Inoltre, un diagramma rende semplice la comprensione del funzionamento di un programma, rispetto alla lettura del codice sorgente e risulta essere utile durante i lavori di sviluppo in team.

In questo caso, dal diagramma della funzione *resource*, che si occupa di gestire le richieste, mostrato nella figura 4.1, si può notare facilmente come il servizio crei un oggetto di tipo client, con riferimento al servizio di API esterno (exchangeAPI) e poi effettui una chiamata *get* su di esso tramite un metodo remoto. Successivamente viene mostrato il flusso logico della funzione, che cambia in base al tipo di risposta che si riceve dal servizio esterno.

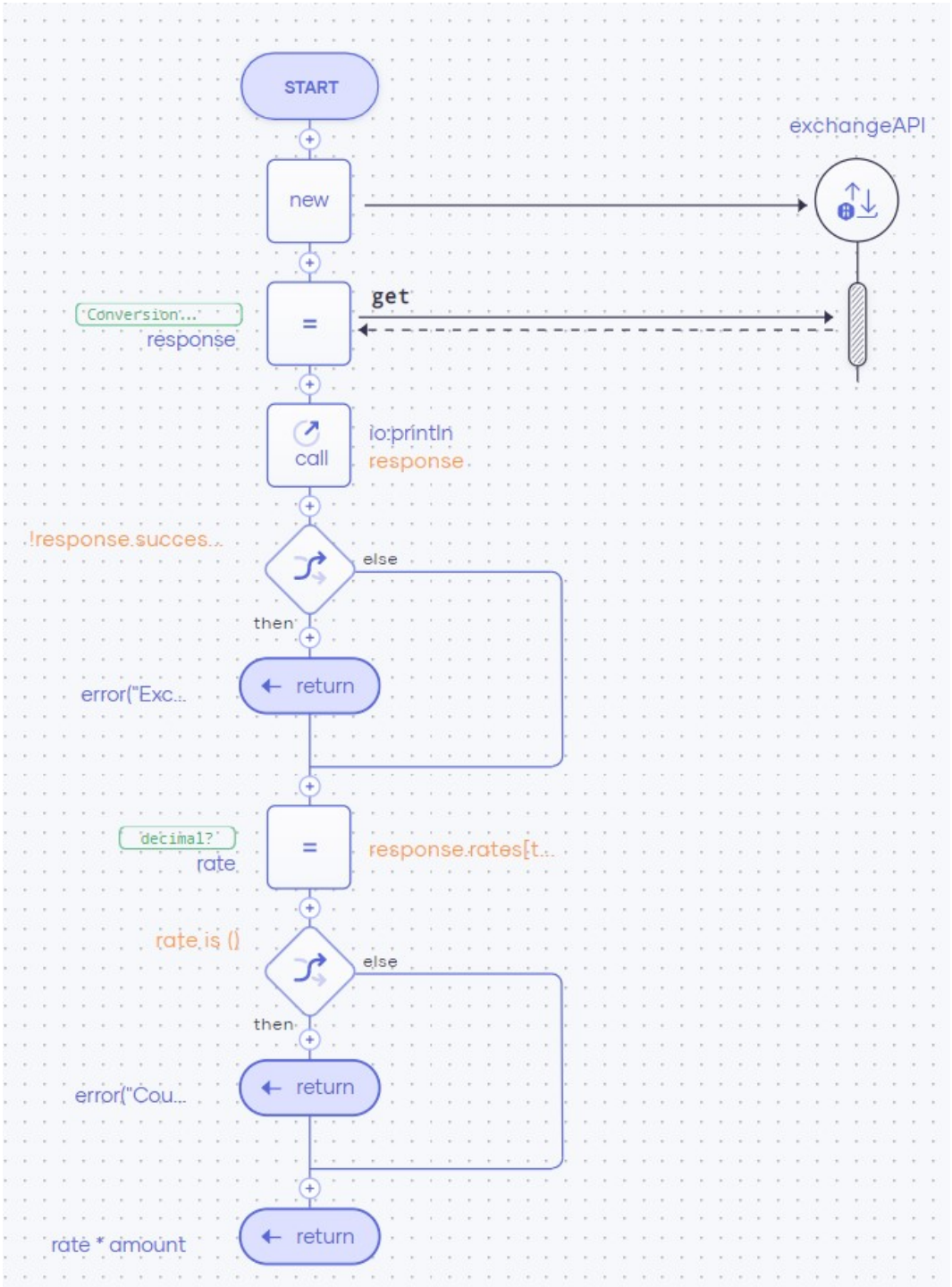

Figura 4.1: Diagramma di sequenza di una funzione del servizio REST

4.6 OpenAPI tool

La specifica OpenAPI [26] è una specifica che crea un **contratto** *RESTful* per le API, dettagliando tutte le sue risorse e operazioni in un formato leggibile dall'uomo e dalla macchina, per facilitare lo sviluppo e l'integrazione. Il tool OpenAPI di Ballerina consente di:

- avviare facilmente lo sviluppo, generando, in Ballerina, il servizio e gli scheletri dei client a partire da un contratto OpenAPI.
- generare le specifiche OpenAPI, relative ad una specifica implementazione di un servizio in Ballerina.
- convalidare la conformità dell'implementazione del servizio rispetto ad un contratto OpenAPI fornito.
- convalidare un'implementazione del servizio rispetto a un contratto OpenAPI durante la fase di compilazione. Questa funzionalità garantisce che l'implementazione di un servizio non si discosti dal suo contratto OpenAPI.

È possibile quindi estrarre le specifiche OpenAPI, dato il servizio creato in precedenza, tramite CLI [26]. Di seguito verrà mostrata una parte del file yaml, contenente le specifiche OpenAPI generate automaticamente tramite il tool di Ballerina.

```
openapi: 3.0.1
info:
  title: RestApi Openapi Yaml
  version: 0.0.0
servers:
- url: "{server}:{port}/"
  variables:
    server:
      default: http://localhost
    port:
      default: port
paths:
  /convert/{base}/to/{target}:
    get:
    ...
```

4.7 Osservabilità

L'osservabilità è una delle funzionalità chiave del linguaggio Ballerina. Essa è una misura di quanto bene gli stati interni di un sistema possono essere dedotti dalla conoscenza dei suoi output esterni. Essa consiste di tre pilastri principali:

- **Metriche**: valori numerici raccolti e aggregati in un periodo di tempo.
- **Tracciamento**: le attività che si verificano quando una richiesta/transazione si verifica nel sistema, dal punto d'ingresso al punto d'uscita.
- **Logging**: record testuali delle attività che si sono verificate, con informazioni rilevanti ed un timestamp associato.

In Ballerina i servizi e tutti i connettori client sono osservabili di default ed il linguaggio consente la completa osservabilità esponendo se stesso, tramite questi tre metodi, a vari sistemi esterni come Prometheus [27], Jeager [28] o Elastic Stack [29].

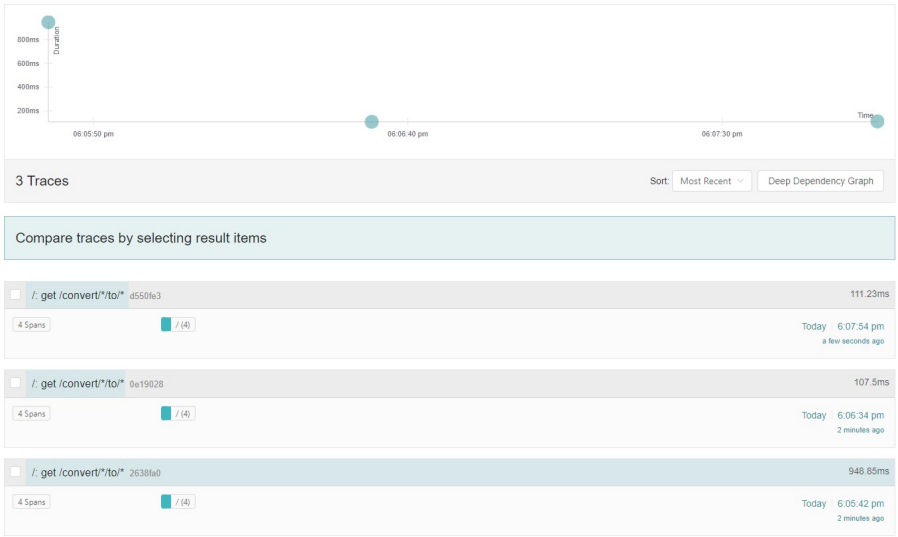

Figura 4.2: Dashboard del servizio Jeager per il tracciamento

In questa figura è possibile osservare la dashboard del servizio Jeager, che permette di effettuare il tracciamento delle attività all'interno del sistema. In questo caso, è possibile osservare come, dopo aver effettuato tre richieste al servizio, esse appaiano all'interno della dashboard, dove è possibile, inoltre, visualizzare grafici o informazioni più dettagliate riguardo le attività del sistema.

Conclusioni

Abbiamo osservato come il linguaggio Ballerina sia stato progettato con l'obiettivo di migliorare l'**integrazione**. Il linguaggio può essere utilizzato per molti scopi nei campi dell'informatica e dell'automazione, tra cui lo sviluppo di applicazioni web, lo sviluppo di API, microservizi, integrazione e ricerca.

I vantaggi principali di questo linguaggio derivano, quindi, da come esso viene utilizzato ed applicato. Uno di questi è che Ballerina, come visto in precedenza, è cloud integrated o **cloud native**. Ciò significa che su progetti complessi la programmazione è preintegrata nel cloud. La pre-integrazione semplifica la collaborazione tra gli sviluppatori e il processo di modifica e manutenzione del codice. Il suo design orientato all'integrazione rappresenta, infatti, una possibile soluzione a molti dei problemi di integrazione, che i progetti di programmazione su larga scala devono affrontare.

Un altro vantaggio offerto da ballerina deriva dalla sua natura **open source**. Il fatto che ballerina sia open source, offre la possibilità a più persone di contribuire al miglioramento del linguaggio. Ciò significa che ballerina può essere sviluppato o adattato in modo efficiente, per soddisfare le esigenze degli utenti. Un aspetto positivo di Ballerina è, sicuramente, quello di essere in continua evoluzione, incrementando il numero di funzionalità di cui dispone. Questa caratteristica può, però, rischiare di essere anche un potenziale lato negativo del linguaggio. Essere uno sviluppatore che lavora con ballerina, richiederà, infatti, di stare al passo con l'evoluzione del linguaggio durante la fase di lavoro, finendo con il complicare il processo di sviluppo di un progetto. Questa complessità aggiuntiva e la necessità di apprendimento del linguaggio, possono potenzialmente scoraggiare le aziende più grandi dall'usare Ballerina, a causa dei tempi e dei costi necessari ad istruire i programmatori [30].

Ballerina è stato il linguaggio di programmazione di punta per la società dello Sri Lanka, WSO2, che lo ha ha inserito negli ideali dell'azienda come progetto a lungo termine, continuando il suo sviluppo. Questo sviluppo verrà, in parte, portato avanti grazie alla sua natura open source. Con l'aumentare dei suoi utenti, aumenterà anche il numero di dati raccolti, che verranno poi utilizzati per migliorare il linguaggio. Ballerina potrebbe, di conseguenza, registrare un'enorme crescita del suo utilizzo a causa della crescente popolarità

dello sviluppo cloud native ed open source.

Il linguaggio è, inoltre, progettato per rimanere **versatile**, al fine di soddisfare le esigenze dei suoi utenti. Potremmo, quindi, vedere ballerina essere usato, sempre più frequentemente, in un'ampia varietà di campi dell'informatica, come la robotica, l'intelligenza artificiale o nell'automazione. Questi campi sono tutti in costante crescita, il che incrementa, ulteriormente, il potenziale di crescita che Ballerina ha come linguaggio di programmazione.

Ringraziamenti

Vorrei, innanzitutto, ringraziare il Prof. Alessandro Ricci, relatore della mia tesi per avermi suggerito l'argomento che ho trattato e per avermi seguito durante tutto il periodo della stesura. Vorrei, poi, ringraziare la mia famiglia, in particolare i miei genitori, che mi hanno sempre supportato e che hanno sempre creduto in me. Infine, un ringraziamento ai miei amici e alle persone che mi sono state vicine durante questo percorso, in particolare, Carlotta, Alex, Alice.

Bibliografia

[1] WSO2. https://wso2.com/about/.

[2] Crystal Bedell. Ballerina Language. https://www.techtarget.com/searchapparchitecture/definition/Ballerina-language.

[3] Dockerfile reference. https://docs.docker.com/engine/reference/builder/.

[4] Kubernetes Documentation. https://kubernetes.io/docs/home/.

[5] The Official YAML Web Site. https://yaml.org/.

[6] Extensible Markup Language (XML). https://www.w3.org/XML/.

[7] Introduzione a JSON. https://www.json.org/json-it.html.

[8] Sameera Jayasoma. Ballerina Runtime — Evolution. https://medium.com/@sameerajayasoma/ballerina-runtime-evolution-f82305e4ab8e, 2017.

[9] What is Java technology and why do I need it? https://www.java.com/en/download/help/whatis_java.html.

[10] Java Virtual Machine (JVM). https://www.w3schools.in/java/java-virtual-machine.

[11] Ballerina - An Open Source JVM Language and Platform for Cloud-Era Application Programmers. https://www.infoq.com/news/2020/01/wso2-releases-ballerina-1-1/.

[12] C Introduction. https://www.w3schools.com/c/c_intro.phpl.

[13] WSO2. Ballerina language specification, 2022R2. https://ballerina.io/spec/lang/2022R2/, 2022.

[14] Technical Quick Start Guide - Unicode. https://home.unicode.org/technical-quick-start-guide/.

[15] WSO2. Ballerina language basics, 2022R2. https://ballerina.io/learn/language-basics/, 2022.

[16] Anjana Fernando and Lakmal Warusawithana. *Beginning Ballerina Programming*. Apress Berkeley, CA, 2020.

[17] Docker overview. https://docs.docker.com/get-started/overview/.

[18] Representational state transfer — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=1121280523, 2022.

[19] HTTP (Hypertext Transfer Protocol). https://www.techtarget.com/whatis/definition/HTTP-Hypertext-Transfer-Protocol.

[20] Introduction to GraphQL. https://graphql.org/learn/.

[21] WSO2. Network interaction. https://ballerina.io/learn/distinctive-language-features/network-interaction/, 2022.

[22] Introduction to SQL. https://www.w3schools.com/sql/sql_intro.asp.

[23] XQuery FLWOR Expressions. https://www.w3schools.com/xml/xquery_flwor.asp.

[24] WSO2. Concurrency. https://ballerina.io/learn/distinctive-language-features/concurrency/, 2022.

[25] X/open xa — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=X/Open_XA&oldid=1116866865, 2022.

[26] command-line interface (CLI). https://www.techtarget.com/searchwindowsserver/definition/command-line-interface-CLI.

[27] Jaeger documentation. https://www.jaegertracing.io/docs/1.39/.

[28] Overview | Prometheus. https://prometheus.io/docs/introduction/overview/.

[29] Elastic Stack: Elasticsearch, Kibana, Beats & Logstash | Elastic. https://www.elastic.co/elastic-stack/.

[30] What is Ballerina. Ballerina Programming Language | by Website Developer | Medium. `https://seattlewebsitedevelopers.medium.com/what-is-ballerina-c6daeaa7a864`.