ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

# IL MODELLO AD ATTORI PER LO SVILUPPO DI APPLICAZIONI WEB ORIENTATE AL CLOUD: IL FRAMEWORK PLAY COME CASO DI STUDIO

Elaborata nel corso di: Sistemi Operativi

*Tesi di Laurea di*:
MATTIA BALDANI

*Relatore*:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2011–2012
SESSIONE II

# PAROLE CHIAVE

**Web**

**Attori**

**Cloud**

**Play**

**Java**

Alla mia famiglia.

# Indice

# Introduzione

Negli ultimi anni le Web application stanno assumendo un ruolo sempre più importante nella vita di ognuno di noi. Se fino a qualche anno fa eravamo abituati ad utilizzare quasi solamente delle applicazioni "native", che venivano eseguite completamente all'interno del nostro Personal Computer, oggi invece molti utenti utilizzano i loro vari dispositivi quasi esclusivamente per accedere a delle Web application. Grazie alle applicazioni Web si sono potuti creare per esempio i social network come Facebook, che sta avendo un enorme successo in tutto il mondo ed ha rivoluzionato il modo di comunicare di molte persone. Inoltre molte applicazioni più tradizionali come le suite per ufficio, sono state trasformate in applicazioni Web come Google Docs, che aggiungono varie funzionalità, tra le quali la possibilità di far lavorare più persone contemporaneamente sullo stesso documento.

Le Web applications stanno assumendo quindi un ruolo sempre più importante, e di conseguenza sta diventando fondamentale poter creare delle applicazioni Web in grado di poter competere con le applicazioni native, che siano quindi in grado di svolgere tutti i compiti che sono stati sempre tradizionalmente svolti dai computer.

In questa Tesi ci proporremo quindi di analizzare le varie possibilità con le quali poter migliorare le applicazioni Web, sia dal punto di vista delle funzioni che esse possono svolgere, sia dal punto di vista della scalabilità. Dato che le applicazioni Web moderne hanno sempre di più la necessità di poter svolgere calcoli in modo concorrente e distribuito, analizzeremo un modello computazionale che si presta particolarmente per progettare questo tipo di software: il modello ad Attori.

Vedremo poi, come caso di studio di framework per la realizzazione di applicazioni Web avanzate, il Play framework: esso si basa sulla piattaforma Akka di programmazione ad Attori, e permette di realizzare in modo semplice applicazioni Web estremamente potenti e scalabili.

Dato che le Web application moderne devono avere già dalla nascita certi requisiti di scalabilità e fault tolerance, affronteremo il problema di come realizzare applicazioni Web predisposte per essere eseguite su piattaforme di Cloud Computing. In particolare vedremo come pubblicare una applicazione Web basata sul Play framework sulla piattaforma Heroku, un servizio di Cloud Computing PaaS.

# Capitolo 1

# Le Web application ed il Cloud Computing

In questo capitolo introdurremo il concetto di Web application e di Cloud Computing, spiegando il come sono nati e quali sono le loro caratteristiche per le quali oggi stanno ricevendo sempre più interesse. In particolare vedremo l'evoluzione delle Web application: i limiti che hanno incontrato dalla loro nascita ed il come si è cercato di risolverli, fino ad arrivare alle nuove tecnologie che ruotano attorno al futuro standard HTML5 ed ai legami con il Cloud Computing, elementi ormai fondamentali delle moderne Web applications.

## 1.1 Le Web applications: che cosa sono e come sono nate

Le Web applications sono sostanzialmente delle applicazioni che utilizzano il World Wide Web come sistema con il quale interagire e comunicare con gli utenti.

A differenza delle applicazioni tradizionali per i computer desktop, che hanno caratterizzato gli anni precedenti all'avvento dell'Internet pervasiva nella quale viviamo oggi, le applicazioni Web non sono un software monolitico che viene eseguito interamente nel calcolatore dell'utente finale, ma al contrario la maggior parte della computazione viene eseguita su un server remoto.

Come tutti sappiamo, infatti, il Web è nato come un sistema distribuito che sfrutta una rete di calcolatori per presentare agli utenti delle informazioni contenute in dei computer centrali, chiamati server. Gli utenti finali per accedere a queste informazioni dovranno utilizzare un software presente sul loro calcolatore (il browser) che riceve le informazioni dal server e le presenta all'utente.

Se però all'inizio il Web è stato concepito semplicemente come sistema per la consultazione di semplici documenti statici come quelli utilizzati dai fisici del CERN, si è successivamente capito che le sue potenzialità erano ben più alte: oltre a documenti statici, potevano essere visualizzati all'utente contenuti generati in quel momento da del software appositamente eseguito sul server. Si incominciarono quindi a creare sistemi come il PHP che permettono di sviluppare del codice che sarà poi eseguito dal server nel momento in cui l'utente ne farà richiesta, ed il cui output sarà inviato e visualizzato nel browser dell'utente.

Fu in questo momento in cui nacque il concetto di Web application: una tipologia di applicazione che viene intesa dall'utente come una semplice pagina Web aperta nel proprio browser, ma che in realtà dietro nasconde una vera e propria applicazione con una certa complessità.

## 1.2 La struttura e le problematiche delle Web applications attuali

Le prime tipologie di Web application che sono state ideate, erano piuttosto basilari e rappresentavano una semplice estensione delle pagine Web statiche utilizzate fino a quel momento. In certi casi si ebbe infatti l'esigenza di aggiungere alle classiche pagine Web delle parti generate dinamicamente dal server, generalmente sulla base di dati presenti su un database. Un esempio di questa tipologia di applicazioni sono i vari CMS (Content Management System), ancora oggi largamente utilizzati per la gestione di siti Web di vario genere, per esempio i blog tramite Wordpress oppure le wiki tramite MediaWiki. La piattaforma più utilizzata per questo genere di applicazioni è stata soprannominata LAMP, dalle iniziali dello stack di software utilizzati lato server: Linux, Apache, MySQL, PHP.

Come è facile intuire da ciò, la parte più importante di queste applicazioni è quella presente lato server: tutti i dati vengono memorizzati in un

database MySQL sul server, dal quale vengono estratti e manipolati da del software, spesso molto basilare, scritto in PHP.

Quando il browser dell'utente richiede la rappresentazione di una certa risorsa ad un server, esso se la farà generare per esempio da degli script PHP, e la invierà in risposta al client che l'ha richiesta, cioè il browser dell'utente. Questo stile architetturale prende il nome di REST, acronimo di Representational State Transfer, ed è quello su cui si basa la maggior parte dei siti e applicazioni Web attuali.

Oltre al software lato server, di cui abbiamo appena parlato, già in tantissime applicazioni Web è in realtà presente del codice che viene eseguito lato client nel browser dell'utente: si tratta degli script scritti nel linguaggio standard JavaScript. Anche se esso è a tutti gli effetti un linguaggio di programmazione adatto teoricamente per qualunque tipo di applicazione, in realtà è stato concepito originariamente come un linguaggio di scripting per effettuare solo semplici operazioni di manipolazione delle pagine HTML all'interno del browser. Se per questo tipo di utilizzo esso è tutto sommato sufficientemente adatto, nel caso invece di software più complesso che deve essere eseguito lato client, esso presenta grosse lacune. Questo rappresenta uno dei problemi delle attuali Web application, in quanto in certi casi occorre necessariamente svolgere lato client alcune computazioni piuttosto complesse, e la mancanza di una piattaforma completa, come può essere Java, per sviluppare codice lato client, rappresenta un grosso handicap che limita le potenzialità di sviluppo delle applicazioni Web.

Negli ultimi anni si stanno affermando sempre di più applicazioni Web molto più complesse di quelle sopra descritte, nelle quali i limiti dati dagli attuali standard JavaScript ed HTML risultano troppo stringenti, fino ad arrivare ad impedirne lo sviluppo. In seguito a questo, si è iniziato lo sviluppo di nuovi standard delle principali tecnologie alla base del Web, che permettano di risolvere i problemi finora incontrati. Questo insieme di nuove tecnologie in fase di sviluppo e standardizzazione, viene comunemente indicato sotto il nome di HTML5, anche se in realtà comprende varie tecnologie diverse tra di loro: per esempio i Web Worker, i WebSocket, l'Offline Storage e nuove versioni del linguaggio JavaScript.

Per risolvere questo ed altri problemi, sono state sviluppate varie tecnologie che verranno racchiuse nello standard HTML5, come i Web Worker ed i WebSocket, che analizzeremo nel prossimo capitolo.

## 1.3 HTML5 e le moderne Web application

Nel mondo delle Web application si sta assistendo negli ultimi anni ad una importante svolta, data dallo sviluppo del futuro standard HTML5 e delle nuove tecnologie ad esso associate. In questo nuovo standard ormai in via di finalizzazione, si sono infatti aggiunte tutta una serie di funzionalità che colmano varie lacune della precedente piattaforma di sviluppo Web, che impedivano alle Web application di svolgere tutte le funzionalità che sono state svolte da sempre dalle tradizionali applicazioni desktop, e più recentemente dalle applicazioni mobile.

Uno dei problemi più evidenti dal punto di vista degli utenti delle Web application attuali, è che nel caso in cui si interrompa la connessione ad Internet, esse smettono di funzionare. Questa è l'ovvia conseguenza del fatto che le attuali tecnologie Web standard non permettono di memorizzare nel browser una sufficiente quantità di informazioni tali da rendere le applicazioni operative anche senza poter scambiare dati con il loro server. Nell'ambito dell'HTML5, si sta cercando di ovviare a questo problema tramite le cosiddette tecniche di Offline Storage, che consentono alle pagine Web di memorizzare delle grandi quantità di dati nel browser dell'utente, per esempio all'interno di un database relazionale.

Un altro problema della parte di software lato client delle Web application attuali, è dato dal modello di computazione utilizzato dal JavaScript tradizionale: dato che esso è nato come linguaggio di scripting per delle semplici manipolazioni del Document Object Model della pagina Web che richiedono pochissima capacità di calcolo, tutto il codice viene eseguito solamente in un unico thread. Di conseguenza non è possibile effettuare delle computazioni che richiedano una quantità di tempo nell'ordine di grandezza dei secondi, perchè esse finirebbero per rendere non responsiva l'intera interfaccia utente.

Per risolvere questo problema sono stati ideati i Web Worker, un sistema grazie al quale è possibile eseguire del codice JavaScript in un thread separato. Per rendere possibile a questi vari thread di comunicare tra di loro, è stato scelto di utilizzare un sistema di message passing che, come vedremo in seguito, può essere modellato utilizzando l'astrazione di Attore.

# Capitolo 2

# Il modello ad Attori per il Web

In questo capitolo analizzeremo nei dettagli il modello ad Attori: vedremo le sue basi teoriche, la sua storia, in particolare i legami con i linguaggi object-oriented, i suoi vantaggi rispetto ad altri modelli computazionali, e le problematiche che lascia irrisolte.

Infine vedremo che vantaggi può portare questo modello nel mondo delle applicazioni Web, dove si ha spesso l'esigenza di realizzare del software ad un elevato grado di concorrenza e distribuito su più nodi.

## 2.1 Le basi del modello ad Attori

Per capire intuitivamente il concetto di Attori, alla luce delle nostre conoscenze della programmazione object-oriented, possiamo pensare per esempio ad un classico oggetto Java al quale viene associato un thread. [1]

Dando una definizione più rigorosa, possiamo definire gli Attori come delle entità che contengono uno stato, e che possono al loro interno eseguire computazioni in completa autonomia dal resto del mondo. Gli oggetti dei linguaggi object-oriented invece sono delle entità "passive", che contengono uno stato e dei metodi, delle istruzioni per modificarlo, ma che autonomamente non sono in grado di svolgere nessuna computazione finchè dall'esterno un flusso di computazione non invoca uno dei loro metodi.

### 2.1.1 I messaggi

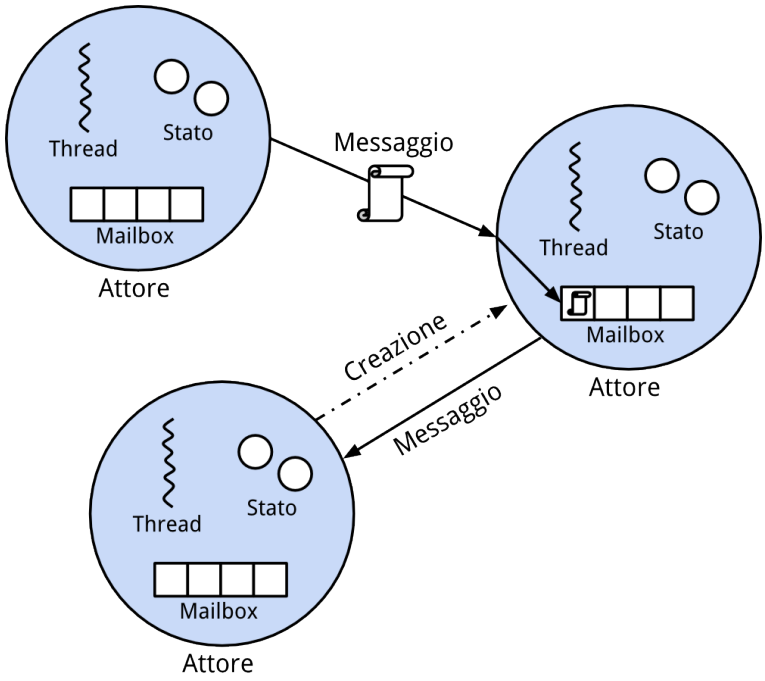
Abbiamo detto che gli Attori possono svolgere al loro interno computazioni in modo completamente autonomo da ciò che hanno all'esterno. Affinchè siano in qualche modo utili, dobbiamo però far si che sia possibile far uscire in qualche modo verso l'esterno le informazioni che contengono.

Nel modello ad Attori ciò avviene tramite i messaggi, una sorta di pacchetti di informazione che un Attore può inviare ad un altro Attore. L'Attore che vuole inviare un messaggio può decidere autonomamente in che momento inviarlo senza nessun vincolo sulla base di ciò che sta succedendo all'esterno, in quanto ha piena autonomia sulle sue azioni.

### 2.1.2 Le mailbox

Un messaggio può quindi in qualsiasi momento partire da un Attore ed arrivare all'Attore destinazione: a questo punto cosa succede? Ovviamente affinchè il messaggio sia utile, esso deve in qualche modo influenzare l'Attore destinazione, però come già detto gli Attori possono svolgere al loro interno computazione in autonomia da ciò che hanno all'esterno, quindi nell'istante di tempo in cui è arrivato il messaggio essi potrebbero essere occupati a svolgere qualcos'altro, quindi non possono riceverlo e svolgere qualcosa in seguito ad esso.

Per risolvere questo problema viene associata ad ogni Attore una mailbox, cioè una sorta di cassetta della posta nella quale vengono depositati i messaggi in arrivo, che in questa analogia vengono considerati come delle buste postali. I vari messaggi in arrivo quindi vengono prima depositati nella mailbox, e successivamente l'Attore, quando ha terminato le sue computazioni interne, può decidere di leggere i messaggi che ha nella propria mailbox, e di comportarsi di conseguenza.




Rappresentazione grafica della struttura interna degli Attori e delle azioni che possono svolgere: invio di un messaggio e creazione di un nuovo Attore.

### 2.1.3 Caratteristiche della comunicazione tra Attori

Come si può facilmente intuire da ciò che è stato appena detto sulle mailbox, la comunicazione tra Attori avviene in modo completamente asincrono. Quando cioè un Attore invia un messaggio ad un altro Attore, esso non ha alcun modo di sapere quando l'Attore ricevente leggerà il messaggio: potrebbe leggerlo immediatamente se per esempio ha la mailbox vuota e non stava svolgendo alcuna azione, potrebbe leggerlo dopo una quantità non ben precisata di tempo perchè per esempio aveva altri messaggi in coda, oppure potrebbe addirittura non leggerlo mai, sia perchè potrebbe per qualche motivo non prelevarlo mai dalla sua mailbox, sia perchè i messaggi potrebbero venire persi durante il transito e quindi non arrivare mai alla mailbox di destinazione.

Per spiegare le differenze tra la comunicazione a messaggi utilizzata dagli Attori e la tradizionale invocazione di metodi propria dei tradizionali linguaggi object-oriented, possiamo fare una analogia con due sistemi

di telecomunicazione che usiamo nella vita di tutti i giorni: gli SMS e le telefonate.

Le telefonate sono un mezzo di comunicazione sincrono, nel quale la voce che un utente trasmette, viene ricevuta immediatamente dall'altro interlocutore. Se esso in quel preciso istante di tempo stava ascoltando allora avrà recepito l'informazione, se invece non stava ascoltando perchè in quel momento era impegnato a svolgere qualcos'altro, avrà perso per sempre l'informazione che gli stava inviando l'altro utente.

Nel caso degli SMS invece, quando un utente riceve un messaggio, non è assolutamente necessario che lo legga immediatamente. Se per esempio l'utente che l'ha ricevuto stava svolgendo qualcos'altro, può leggere l'SMS successivamente senza perdere alcuna informazione. In questo caso però l'utente che trasmette il messaggio non potrà mai sapere quando il destinatario leggerà il messaggio, e non ha neanche alcuna garanzia che l'abbia effettivamente ricevuto.

Da questa analogia possiamo anche spiegare le differenze tra il modello ad Attori e quello dei tradizionali linguaggi object-oriented: mentre un messaggio scambiato tra Attori può essere equiparato ad un SMS, una invocazione di un metodo di un oggetto può essere considerata come una specie di telefonata. Quando infatti viene invocato un metodo, il flusso di computazione viene completamente dedicato ad esso, finchè l'esecuzione di quel metodo non termina.

### 2.1.4 Il comportamento degli Attori

Abbiamo quindi compreso in che modo un Attore può comunicare e scambiare informazioni con il mondo esterno. Ora vediamo invece più in dettaglio come funziona un Attore internamente, cioè che azioni può svolgere e sulla base di cosa sceglie che azione svolgere.

Come abbiamo già detto un Attore può svolgere al suo interno una computazione in modo completamente indipendente dall'esterno: in questo caso la scelta di che azione svolgere può dipendere necessariamente solo dal suo stato interno, cioè da ciò che in ambito object-oriented definiremmo come campi dell'oggetto, poichè non ha alcun altro grado di libertà da cui attingere informazioni. Nel caso invece in cui l'Attore riceve un messaggio dall'esterno, esso potrà utilizzare anche le informazioni contenute nel messaggio per decidere le proprie azioni: come nel caso della programmazione

funzionale, dove variando il valore dei parametri di una funzione si può ottenere un diverso valore di ritorno, anche nel caso degli Attori variando le informazioni contenuti nei messaggi ricevuti si può influenzare il risultato della loro computazione.

Un modo fondamentale per descrivere e caratterizzare un Attore dovrà quindi essere il suo modo di reagire ai messaggi che riceve, che in analogia con gli esseri viventi possiamo definire come comportamento.

Ora vediamo quali sono le conseguenze che può causare una certa computazione all'interno di un Attore, cioè quali sono le azioni che un Attore può svolgere. Abbiamo già detto che un Attore deve poter modificare il suo stato interno ed inviare messaggi ad altri Attori. L'ultima operazione fondamentale della quale non abbiamo ancora parlato, è la creazione di un nuovo Attore. I dettagli di questa operazione non sono ben specificati dal modello generale di Attore, sopratutto per ciò che riguarda le proprietà e lo stato in cui verrà a trovarsi l'Attore appena creato. In certi framework di programmazione ad Attori, questo viene definito specificando la classe, nel senso object-oriented del termine, sulla base della quale verrà creato il nuovo Attore. Questo comunque non è importante per ciò che riguarda il modello di Attore; ciò che ci interessa sapere è che un Attore può in qualsiasi momento svolgere l'azione di creare un nuovo Attore, indipendentemente da ciò che succede all'esterno di esso.

## 2.2 I vantaggi del modello ad Attori nella programmazione concorrente e distribuita

Il modello ad Attori che abbiamo precedentemente esposto, è particolarmente utile nel caso della progettazione di applicazioni distribuite che fanno uso di un elevato grado di concorrenza. Le applicazioni Web moderne basati sull'HTML5 e le tecnologie ad esso correlate, rientrano in questa categoria, in quanto devono poter svolgere computazioni distribuite su vari calcolatori della rete, quindi intrinsecamente concorrenti.

### 2.2.1 Software concorrente ad Attori

Tramite gli Attori si possono risolvere in modo elegante i classici problemi dati dalla concorrenza, che tradizionalmente rendevano la programmazione concorrente piuttosto complessa e soggetta ad errori imprevedibili. Utilizzando infatti la più tradizionale astrazione di thread e di memoria condivisa, è necessario ricorrere a tecniche di locking per evitare delle corse critiche: queste tecniche sono spesso molto poco intuitive, e rendono piuttosto difficile la progettazione del software. In certi casi inoltre si è portati ad utilizzare i lock in maniera non ottimale, generando anche dei problemi di prestazioni dato dall'utilizzo non efficiente di tutti i processori messi a disposizione.

Utilizzando il modello ad Attori questi problemi vengono risolti progettando il proprio sistema software in modo completamente differente: invece di utilizzare della memoria condivisa che renderebbe necessarie le tecniche di locking, viene utilizzato il message passing tra vari Attori.

Il modello ad Attori è un modo quindi particolarmente semplice e comodo per progettare applicazioni molto concorrenti, come possono essere per esempio i Webserver e le applicazioni Web che vengono eseguite su di essi, che devono arrivare a gestire molte migliaia di richieste contemporanee. Nei Webserver tradizionali come l'Apache HTTP Server, viene infatti associato un thread od un processo per ogni richiesta HTTP in fase di elaborazione, poichè vengono effettuate delle read e write bloccanti sui socket TCP. Ciò crea molti problemi di prestazioni e di utilizzo di memoria RAM, limitando il numero di richieste contemporanee che il Webserver può gestire. Questo problema è stato risolto in Webserver come nginx [**?**] utilizzando un sistema basato sugli eventi [**?**], simile all'event-loop utilizzato nel framework ad Attori Akka che vedremo nel prossimo capitolo.

Nei capitoli successivi vedremo più in dettaglio dei possibili utilizzi del modello ad Attori per la realizzazione applicazioni server-side concorrenti, ora invece vediamo brevemente come utilizzarlo per realizzare il software lato client delle applicazioni Web, tramite i Web Worker.

### 2.2.2 I Web Worker: programmazione JavaScript ad Attori

Nelle Web application moderne, come già detto, si sta avendo sempre più spesso la necessità di svolgere delle computazioni piuttosto complesse anche

lato client, nel browser dell'utente. Con gli standard Web attuali, questo tipo di software può essere realizzato solo in JavaScript, con il suo classico modello computazionale nel quale tutto il codice viene eseguito in un solo thread. Quindi non è possibile eseguire calcoli che richiedano molto tempo, in quanto bloccherebbero il resto dell'interfaccia utente della pagina HTML.

Negli nuovi standard Web in via di sviluppo è stata invece introdotta una tecnologia che permette di eseguire calcoli complessi anche lato client: i Web Worker. Come si intuisce dal nome, essi permettono di eseguire della computazione in un flusso di esecuzione separato da quello principale nel quale viene eseguito il resto del codice JavaScript della pagina Web.

Per scambiare dati tra i vari di worker, ed anche per dare inizio alla loro computazione, viene utilizzato un sistema di message passing: eseguendo il metodo postMessage su di un oggetto che rappresenta un worker, gli viene inviato un messaggio, che causerà in esso l'esecuzione di un handler dell'evento di ricezione del messaggio.

In questo esempio di codice viene creato un worker, al quale viene inviato un messaggio, e viene settato un handler degli eventuali messaggi di risposta da parte del worker:

```
var worker = new Worker('doWork.js');

worker.addEventListener('message', function(e) {
  console.log('Worker said: ', e.data);
}, false);

worker.postMessage('Hello World');
```

Questo sistema di message passing, alla luce delle nostre conoscenze sul modello ad Attori, può essere interpretato in questo modo: i worker vengono rappresentati come degli Attori, che si scambiano dei messaggi asincroni come descritti nel precedente capitolo.

Tramite il modello ad Attori è stato quindi possibile realizzare una tecnologia come i Web Worker che permette di realizzare script concorrenti eseguiti da un browser Web senza utilizzare le tradizionali tecniche di locking ed i problemi da esse derivati, come le possibili corse critiche.

### 2.2.3 Analogie fra message passing tra Attori, e comunicazioni tra nodi della rete

Nel modello ad Attori, i messaggi scambiati tra i vari Attori sono una sorta di buste che contenenti dei dati che vengono depositate nella mailbox dell'Attore destinatario. Da questo concetto di messaggi del modello ad Attori, si può intuire una certa analogia con le reti di calcolatori: la comunicazione tra due computer via rete avviene tramite la trasmissione di dati che generalmente avviene in modo asincrono con il flusso di esecuzione del software di ognuno dei due computer. Analogamente, nel caso del modello ad Attori, i messaggi sono asincroni, e possono essere letti dall'Attore destinatario in un istante di tempo successivo da quello in cui sono stati inviati dall'Attore mittente. Potremmo quindi considerare i vari nodi della rete come degli Attori, ed i messaggi tra essi trasmessi come dei messaggi scambiati tra questi Attori.

Lo scambio di messaggi asincroni come quelli del modello ad Attori, può essere molto utile in molti ambiti, tra i quali anche quello delle applicazioni Web: per esempio in una Web application di chat tra vari utenti, i messaggi di chat sono una sorta di messaggi asincroni che devono essere inviati dal browser dell'utente mittente al server Web, e successivamente dal server Web al browser dell'utente destinatario. Tramite il protocollo HTTP ed il modello REST però non è possibile scambiare messaggi in modo bidirezionale tra i server Web ed i browser: per questo motivo tra le tecnologie Web legate al futuro standard HTML5, si è ideato il meccanismo dei WebSocket, che permette di scambiare dati in modo bidirezionale in modo analogo a quanto avviene nei tradizionali socket di rete connection-oriented dei sistemi operativi.

# Capitolo 3

# Framework ad attori

In questo capitolo analizzeremo i vari framework utilizzabili per realizzare del software progettato con il modello ad Attori. Vedremo brevemente la storia dei linguaggi e framework che hanno preso ispirazione da esso, e poi analizzeremo nei dettagli quelli più recenti ed utilizzati. In particolare ci soffermeremo su quelli basati sulla JVM, prendendo come esempio il framework Akka.

## 3.1 Storia delle implementazioni del modello ad Attori

Il modello ad Attori fu formulato dal punto di vista matematico in alcune pubblicazioni scientifiche del 1973. Da quel momento molti gruppi di ricerca iniziarono a sviluppare vari linguaggi di programmazione e piattaforme che si ispirassero ad esso. Mentre nei primi anni questa ricerca fu svolta solo in ambito accademico, in quanto a quel tempo non si sentiva ancora la necessità nel mondo dell'industria di avere sistemi di computazione concorrente e distribuita, successivamente con l'avvento dei processori multicore e di Internet aumentò l'interesse nell'utilizzare il modello ad Attori anche in applicazioni reali.

Il primo linguaggio realizzato ed utilizzato in ambito industriale che implementa i concetti del modello ad Attori, fu l'Erlang, ideato dalla Ericsonn nel 1986 per essere utilizzato nei propri switch ed altri apparati di rete che necessitano di essere sempre attivi e tolleranti ai guasti. Il linguaggio Erlang

ebbe un grande successo nel settore delle telecomunicazioni, in quanto in questo ambito è importantissimo avere sistemi con una alta availability.

Successivamente il modello ad Attori fu implementato per vari linguaggi più general purpose come il Java, con lo scopo di realizzare in modo semplice del software con computazioni concorrenti ed eventualmente distribuite su vari calcolatori, esigenze sempre più sentite con l'avvento di processori con un numero di core sempre più alto, e delle reti ad alta velocità. Delle librerie per la programmazione ad Attori sono state realizzate molti linguaggi tra i quali: C, C++, Python, C#, Java e Scala.

Tra queste tante soluzioni, negli ultimi anni si stanno diffondendo sempre di più quelle basate sulle piattaforme Java e .NET. Queste due piattaforme infatti sono tra le più utilizzate in ambito enterprise, e sono già utilizzate come base per una grandissima quantità di software. Diventa quindi naturale cercare di utilizzare il modello ad Attori in queste piattaforme già esistenti e molto diffuse, in modo da evitare di dover riscrivere interamente del codice incompatibile con quello vecchio. Tra queste due, la piattaforma più diffusa e più importante da analizzare è Java, in quanto sulla JVM, oltre al linguaggio Java, sono stati realizzati anche altri linguaggi che possono essere eseguiti su essa, per esempio Scala che è particolarmente adatto per la programmazione ad Attori.

Il linguaggio Scala è stato ideato nel 2003 da Martin Odersky, il quale ha poi fondato l'azienda Typesafe, che si occupa dello sviluppo e del supporto sul linguaggio Scala. Oltre a questo la Typesafe ha poi sviluppato Akka, un framework con un quale si può realizzare facilmente del software concorrente e distribuito progettato con il modello ad Attori, utilizzando i linguaggi Java o Scala.

## 3.2 Framework e librerie basati sulla JVM

Come abbiamo già detto, le soluzioni per la programmazione ad Attori più interessanti da analizzare sono quelle che possono essere eseguite sulla JVM, in quanto è già disponibile una grande di quantità di software in ambito enterprise basato sulla piattaforma Java. Restando nell'ambiente Java quindi, il nuovo codice che si andrà a sviluppare utilizzando il modello ad Attori, potrà essere perfettamente integrato con il software legacy preesistente.

Sulla JVM è possibile eseguire del codice bytecode prodotto dalla compilazione di vari linguaggi di programmazione: tra questi i due più importanti sono ovviamente Java, e Scala.

Prima che si diffondesse il linguaggio Scala, furono realizzate diverse librerie per il linguaggio Java che semplifichino l'implementazione di software utilizzante il modello ad Attori, per esempio ActorFoundry. [2] Però a causa di alcune caratteristiche intrinseche del linguaggio Java, scrivendo del software ad Attori per questo linguaggio, si ottiene un codice leggermente "sporco". Utilizzando Scala invece, grazie a delle caratteristiche avanzate del linguaggio stesso, è possibile scrivere software ad Attori in modo più semplice ed elegante. Per questo motivo la programmazione ad Attori sui linguaggi JVM-based ha iniziato ad avere veramente successo con la diffusione di Scala.

Tra i framework di questo tipo, il più recente ed utilizzato è Akka. Esso è stato ideato per il linguaggio Scala, e quindi può sfruttare pienamente tutte le sue feature più avanzate, ma parallelamente permette di sviluppare codice in Java. Si tratta quindi di un framework molto vantaggioso da utilizzare in ambito industriale, in quanto unisce le feature più avanzate di Scala, con la retrocompatibilità e l'enorme diffusione di Java.

## 3.3 Il framework Akka


Akka è un framework ideato per realizzare facilmente applicazione distribuite e con un elevato grado di concorrenza. [3] Essendo stato realizzato dalla Typesafe, l'azienda fondata dall'ideatore di Scala, è stato progettato alla luce di tutte le feature più avanzate del linguaggio Scala, ed è il

framework consigliato dall'azienda stessa per chi vuole scrivere software in Scala utilizzando il modello ad Attori. Come già detto però Akka permette allo stesso tempo permette di scrivere codice anche in linguaggio Java, senza perdere nessuna delle funzionalità più importanti offerte dal framework. In questo modo chi conosce il linguaggio Scala e deve scrivere del codice ex-novo può utilizzare Scala, mentre chi conosce solo il linguaggio Java oppure chi ha la necessità di fare un porting di codice Java già esistente, può utilizzare tranquillamente il linguaggio Java senza rinunciare alle potenzialità di Akka.

Sviluppando software ad Attori in Akka, oltre all'evidente vantaggio di poter produrre facilmente del software concorrente senza dover tecniche di locking, si hanno altri grossi vantaggi dati dalle funzionalità di remoting e di fault tolerance: una volta realizzata un applicazione in Akka, si può decidere anche a runtime di spostare parte degli Attori che la compongono in un altro computer, in modo completamente trasparente al resto dell'applicazione. Inoltre, se l'applicazione è realizzata correttamente, dovrebbe poter tollerare un malfunzionamento di ogni sua parte, grazie alle funzioni di gracefully failure che spiegheremo più avanti.

Nei paragrafi seguenti, vedremo il funzionamento di Akka utilizzando nelle esempi di codice solo le API per il linguaggio Java, tuttavia è possibile passare con molta facilità alle corrispondenti API Scala, dato che sono state progettate per essere molto simili tra di loro.

### 3.3.1 L'implementazione degli Attori Akka

In Akka ogni attore viene rappresentato da una classe Java che estende la classe UntypedActor, e deve di conseguenza implementare il metodo onReceive, che è una sorta di handler della ricezione di un messaggio da parte di un Attore generato sulla base di quella classe. [4]

Di seguito vediamo un esempio di Attore Akka scritto in Java, che svolge un compito molto semplice: quando esso riceve un messaggio, controlla se esso è di tipo String, ed in questo caso stampa la stringa sullo standard output:

```
public class PrintStringActor extends UntypedActor {
    public void onReceive(Object message) throws Exception {
        if (message instanceof String){
```

```
            System.out.println(\Messaggio stringa ricevuto: \ + message);
        } else {
            unhandled(message);
        }
    }
}
```

Quando un messaggio arriva ad un Attore, il sistema Akka si occupa di invocare questo metodo onReceive passandogli come argomento un oggetto che rappresenta il messaggio. I messaggi che si scambiano i vari attori Akka sono dei oggetti Java (POJO) senza dei vincoli ben precisi, l'unica regola che devono rispettare è che siano immutabili, altrimenti un messaggio potrebbe essere in grado di variare il suo stato mentre transita da un Attore ed un altro, e questo violerebbe completamente le regole stabilite dal modello ad Attori. Gli oggetti che rappresentano i messaggi inoltre vengono passati al metodo onReceive come oggetti di tipo Object, quindi nell'implementazione del metodo onReceive dovremmo controllare di che tipo sono realmente i messaggi, per esempio con la keywork instanceof.

Quando implementiamo un Attore in Akka dobbiamo quindi stabilire il codice che deve essere eseguito all'arrivo di ogni tipo di messaggio. Nel fare ciò dobbiamo ricordarci che nulla vieta che ad un Attore venga trasmesso un messaggio di un tipo diverso da quelli per cui è stato progettato di ricevere, quindi dobbiamo decidere anche che codice eseguire nel caso arrivi un messaggio di tipo sconosciuto. Generalmente in questo caso viene chiamato il metodo unhandled messo a disposizione dal sistema Akka.

All'arrivo di un certo tipo di messaggio, il nostro Attore dovrà quindi svolgere una certa azione: Akka ci permette di fargli eseguire del codice Java arbitrario, tuttavia dobbiamo di rispettare rispettare alcune regole per restare all'interno del modello ad Attori. La regola più importante è di non effettuare delle operazioni di input/output bloccanti come la lettura di file su disco o dalla rete, ma solo operazioni che sfruttino pienamente la CPU per tutto il tempo in cui sono in esecuzione. Per capire il perchè di ciò, vediamo più in profondita come funziona internamente il sistema ad Attori di Akka.

### 3.3.2 Elaborazione dei messaggi asincroni tramite gli event-loop

In Akka, per eseguire il codice nel metodo onReceive dei vari Attori, viene fatto uso di un thread-pool, un insieme di thread creati e gestiti dal framework Akka, ai quali vengono "agganciati" man mano i frammenti di codice da eseguire. Generalmente il numero di thread del thread-pool viene fatto corrispondere al numero di processori (logici) della macchina sulla quale si sta eseguendo la JVM, in modo che sia possibile avere un flusso di esecuzione per ognuno di essi quando necessario.

Il sistema funziona in questo modo: su ogni thread viene messo in esecuzione un worker, una sorta di loop infinito nel quale ad ogni ciclo si cerca di prendere del lavoro da eseguire da una coda, e lo si esegue. L'operazione di get del lavoro da una coda è bloccante, in quanto potrebbe non esserci lavoro da svolgere. In questo caso il thread resta in idle senza fare nulla finchè non viene messo in coda del lavoro da svolgere. Il lavoro da svolgere non è altro che l'esecuzione del codice nel metodo onReceive di un certo Attore quando esso riceve un messaggio. Siccome l'arrivo di un messaggio ad un Attore può essere considerato come una sorta di evento, questa tecnica viene chiamata event-loop.

Da qui si intuisce perchè il codice nell'onReceive degli Attore deve essere non bloccante: se esso rimane per esempio in stato di iowait per 10 secondi, esso occuperà un thread del thread-pool per almeno 10 secondi, anche se in realtà non sta utilizzando affatto la CPU. In questo modo quell'Attore occupa inutilmente un thread che invece potrebbe essere usato per svolgere del lavoro CPU-bound da parte di un altro Attore.

Questa tecnica degli event-loop viene utilizzata anche in molti altri software, per esempio in nginx, un webserver progettato per reggere un numero enorme di connessioni contemporanee. Questo è possibile in quanto nei webserver tradizionali, ad ogni richiesta veniva associato un thread o addirittura un processo, e ciò ha un costo computazionale e di memoria molto elevato, anche se per la maggior parte del tempo questi thread restano in stato di wait. Utilizzando invece un sistema con un event-loop, si può gestire un numero enorme di connessioni con pochissimi thread, i quali verranno occupati solo negli istanti in cui accade un certo evento in seguito al quale va svolto del lavoro CPU-bound.

### 3.3.3 Gestione degli errori: gracefully failure

Un altra feature molto importante di Akka è la sua gestione delle eccezioni, che viene definita dai suoi autori come "Gracefully failure". Questa definizione sta a significare che nel caso che una parte dell'applicazione fallisca nello svolgere un certo lavoro, il resto dell'applicazione rimane comunque in grado di funzionare, eventualmente cercando di far ripartire di nuovo l'operazione fallita.

Il modello ad Attori ci viene in aiuto nel realizzare questa feature, in quanto siccome i messaggi scambiati tra gli Attori sono asincroni, non vi è alcuna garanzia sul tempo massimo che essi ci possano impiegare per arrivare a destinazione, questo tempo potrebbe essere talmente alto da essere considerato infinito. Quando si sviluppa un applicazione ad Attori bisogna quindi essere coscienti che i messaggi che vengono inviati ad un Attore, potrebbero non arrivargli mai, ed ovviamente l'applicazione deve essere in grado di tollerare il fatto che un messaggio non arrivi a destinazione. In Akka questo viene effettuato in modo semplice dando a certi Attori il ruolo di supervisori di altri Attori. In sostanza il supervisore deve controllare che l'Attore sotto la sua supervisione stia funzionando correttamente, e nel caso si accorga di qualche problema deve agire di conseguenza per esempio riavviando l'Attore malfunzionante oppure reinviandogli di nuovo un certo messaggio. Generalmente quando viene inviata ad un Attore una certa richiesta di svolgimento di un lavoro, viene fissato un timeout oltre al quale, se l'Attore non risponde con il risultato, l'Attore viene considerato malfunzionante.

### 3.3.4 Il Remoting

L'ultima feature molto importante di Akka che vedremo è il Remoting, che permette di realizzare applicazioni distribuite in una rete di calcolatori in modo molto semplice e robusto. In pratica dopo aver sviluppato normalmente una certa applicazione, si può decidere anche a runtime di spostare una parte dei suoi Attori su un altro computer. [5] Dato che i messaggi asincroni del modello ad Attori hanno le stesse caratteristiche dei pacchetti di una rete di computer, è molto immediato distribuire gli Attori in rete: basta fare in modo che i messaggi che devono essere scambiati tra due Attori che sono in due diversi computer, vengano serializzati ed inviati via rete all'altro computer.

Grazie alle feature già viste di gracefully failure, realizzando applicazioni ad Attori in Akka si ha, senza quasi alcuno sforzo aggiuntivo, il vantaggio di poter distribuire il lavoro su più computer in modo trasparente e tollerante ai guasti.

Questo è molto importante in uno scenario di Cloud Computing, in quanto è possibile spostare dinamicamente a runtime delle intere parti di applicazione da un server all'altro della rete Internet, in modo completamente trasparente e resistente ai guasti.

# Capitolo 4

# Caso di studio: il Play framework


In questo capitolo analizzeremo Play, un framework per lo sviluppo di applicazioni Web moderne basato sul framework Akka di programmazione ad Attori, visto nel capitolo precedente.

Vedremo come funziona internamente il Play framework, che vantaggi ha rispetto ad alcuni altri framework più recenti per lo sviluppo Web, ed infine vedremo come sviluppare un'applicazione Web utilizzando il framework Play.

Durante la spiegazione di questi aspetti, evidenzieremo sopratutto i vantaggi che ha dato la scelta di utilizzare il modello ad Attori nel Play framework, ed anche ciò che rende Play particolarmente adatto per realizzare applicazioni scalabili pensate per funzionare in ambiente Cloud [**?**].

## 4.1 Funzionalità principali e struttura del Play framework

### 4.1.1 Struttura generale del framework

Il Play framework é sostanzialmente una libreria di codice basato su Akka che fornisce tutte le funzionalità necessarie per sviluppare un'applicazione Web e metterla in esecuzione lato server.

Play è stato scritto completamente in Scala, utilizzando Akka ed il modello ad Attori, tuttavia da allo sviluppatore Web la possibilità di utilizzare sia il linguaggio Scala sia il linguaggio Java, in modo analogo ad Akka.

Un'applicazione Web realizzata con Play é a tutti gli effetti un software utilizzante il modello ad Attori implementato su Akka, tuttavia Play è stato progettato in modo da rendere semplice ed intuitivo lo sviluppo di applicazioni Web anche agli sviluppatori che non conoscono le fondamenta del modello ad Attori. Lo sviluppo di applicazioni su Play segue infatti il semplice e consueto pattern Model-View-Controller, al quale molti sviluppatori sono già abituati dato che esso è utilizzato in molti altri framework per lo sviluppo Web, per esempio Ruby on Rails o Symphony.

Il Play framework integra già dentro di se il server HTTP Netty, opportunamente adattato per essere perfettamente integrato con il resto della piattaforma basata su Akka. Esso supporta una gestione completamente asincrona ad eventi del trasferimento di dati, come già visto nel capito precedente riguardo ad Akka.

### 4.1.2 Struttura di una tipica applicazione Play

Vediamo ora un esempio di come è strutturato il codice di una tipica applicazione Play. Esso viene diviso per comodità in varie directory, che formano questa struttura ad albero:

```
app                      --- Application sources
 \__ assets                --- Compiled asset sources
    \__ stylesheets        --- Typically LESS CSS sources
    \__ javascripts        --- Typically CoffeeScript sources
 \__ controllers           --- Application controllers
```

```
 \__ models                  --- Application business layer
 \__ views                   --- Templates

conf                        --- Configurations files
 \__ application.conf        --- Main configuration file
 \__ routes                  --- Routes definition

public                      --- Public assets
 \__ stylesheets             --- CSS files
 \__ javascripts             --- Javascript files
 \__ images                  --- Image files

project                     --- sbt configuration files
 \__ build.properties        --- Marker for sbt project
 \__ Build.scala             --- Application build script
 \__ plugins.sbt             --- sbt plugins

lib                         --- Unmanaged libraries dependencies

logs                        --- Standard logs folder
 \__ application.log         --- Default log file

target                      --- Generated stuff
 \__ scala-2.9.1
    \__ cache
    \__ classes              --- Compiled class files
    \__ classes_managed      --- Managed class files (templates, ...)
    \__ resource_managed     --- Managed resources (less, ...)
    \__ src_managed          --- Generated sources (templates, ...)

test                        --- unit and functional tests
```

La directory dove è presente il codice vero e proprio dell'applicazione è "app": al suo interno troviamo un ulteriore suddivisione effettuata sulla base del modello Model-View-Controller, sul quale il Play framework si basa.

Il codice che viene eseguito appena arriva al webserver una richiesta HTTP, è quello presente nella directory "controllers": in essa vanno inseriti

dei file sorgenti Java contenenti ognuno una classe pubblica che estende Controller. In queste classi vanno inseriti una serie di metodi statici, ognuno dei quali rappresenta un certo tipo di "azione" che può essere effettuata sul server: essi dovranno accettare come argomenti i dati necessari all'esecuzione dell'azione che dovranno essere inviati al server tramite la corrispondente richiesta HTTP, per esempio inviandoli come una Entity HTTP utilizzando il metodo POST, oppure tramite il metodo GET includendoli nell'URL come parametri. Una volta terminata l'esecuzione di questo metodo, il framework Play invierà al client i dati contenuti nel valore di ritorno del metodo, che costituiranno quindi la Entity HTTP di risposta alla richiesta HTTP, che nella maggior parte dei casi sarà una semplice pagina HTML.

Un semplice esempio di metodo di una classe del "controller" è questo, che ritorna al client del semplice testo contenente il nome che gli è stato passato come parametro:

```
public static Result hello(String name) {
    return ok("Hello" + name);
}
```

Quando viene effettuata una richiesta HTTP al Webserver di Play, esso cercherà sulla base dei dati contenuti nel file "conf/routes", qual'è il metodo delle classi "controllers" corrispondente alla URL specificata nella richiesta HTTP inviata dal client. Nel file "routes" è possibile associare ad ogni metodo uno o più pattern di URL, che vengono confrontati con un sistema di matching con la URL di richiesta inviata dal client. Per accedere alla nostra azione di esempio, potremmo utilizzare questa route:

```
GET   /hello/*name     controllers.Application.hello(name)
```

Di conseguenza, se un utente accederà a questa URL:

```
http://www.miodominio.com/hello/MattiaBaldani
```

riceverà questo testo in risposta:

```
Hello MattiaBaldani
```

Tramite questo sistema, è possibile creare per la propria applicazione Web delle URL completamente personalizzate, in modo analogo a ciò che viene effettuato con il modulo "mod_rewrite" dell'Apache Web Server. Questo è particolamente utile nelle Web application complesse per avere delle URL più "pulite", cioè più facilmente leggibili ed interpretabili dall'utente finale e dai motori di ricerca, che in molti casi premiano i siti con URL pulite posizionandoli più in alto nei risultati delle proprie ricerche.

Chi naviga spesso su vari siti Web, si sarà frequentemente imbattuto in URL di questo tipo:

```
http://example.com/cgi-bin/index.jsp?cat=2&id=medical%20patents
                                          &ref=a&sec=services
```

Queste URL, anche se sono perfettamente aderenti agli standard Web, hanno uno scarso contenuto informativo per l'utente finale, poichè esso leggendole non capisce il loro significato. Se invece questa URL fosse riscritta in questo modo, sarebbe sicuramente molto più comprensibile ed utile agli utenti:

```
http://example.com/services/legal/medical_patents
```

Ritornando alla struttura delle applicazioni Play, vediamo a cosa servono le altre directory contenute in "app":

Nella directory "models" vanno inseriti dei file sorgenti Java delle classi che rappresentano gli oggetti ed i dati del proprio dominio applicativo. Queste possono essere delle semplici classi Java POJO che verranno create per esempio da un metodo del controller, oppure può essere effettuato un mapping tra una di queste classi ed una entità di un database. Per questo fine, Play integra la libreria Ebean di Object-Relational Mapping (ORM) [8], che permette di leggere e scrivere dei dati da un database relazionale grazie alle classi del model [9]. Se per esempio abbiamo un database MySQL che contiene questa tabella degli utenti:

```
Utenti(idUtente, nome, cognome, email)
```

possiamo creare questa classe Java nella directory "models":

```
@Entity
public class Utente extends Model {

    @Id
    @Constraints.Min(10)
    public Long idUtente;

    public String nome;

    public String cognome;

    @Constraints.Required
    public String email;

    public static Finder<Long,Task> find =
            new Finder<Long,Task>(Long.class, Task.class);

}
```

Invocando dei metodi dell'oggetto Finder find, è possibile eseguire delle query sul database relazionale: per esempio è possibile far creare automaticamente tramite libreria Ebean, degli oggetti Java di classe Utente contenenti nei vari campi i corrispondenti valori di un record della tabella Utenti del database MySQL.

Grazie a questa libreria di ORM, è possibile accedere ad un database relazionale dalla propria applicazione Play in modo molto semplice senza la necessità di scrivere delle query in linguaggio SQL, evitando tra l'altro dei possibili bug di sicurezza di tipo SQL Injection, che sono veramente molto comuni per esempio nelle applicazioni Web scritte in PHP. Inoltre, siccome i dati devono essere inseriti in un oggetto Java, si ha un controllo di tipo dei dati prelevati o scritti sul database, in modo da evitare errori legati al typing dei dati.

Nella directory "views" vanno invece inseriti dei file che contengono il template HTML delle pagine della propria applicazione Web. In questi file di testo va scritto del codice HTML tra il quale vanno inseriti degli appositi tag del tipo @( codiceScala ), al posto dei quali Play sostituirà il valore di ritorno della funzione scritta in linguaggio Scala presente in codiceScala.

### 4.1.3 Scalabilità: le variabili di sessione lato client e le richieste stateless

Play è stato progettato per lo sviluppo di applicazioni ad alta scalabilità, cioè che possano essere distribuite a runtime su più server in caso di necessità, in modo da poter servire un numero di utenti teoricamente illimitato semplicemente aggiungendo altri server.

Per far si che ciò sia possibile, occorre che le richieste HTTP provenienti dai vari client, siano spartite tra i vari server, e che ognuna di esse interessi solo il server a cui è stata diretta. In molti framework Web tradizionali come JSF o Spring ed in linguaggi come il PHP, questa regola non viene rispettata, poichè essi devono offrire allo sviluppatore la possibilità di memorizzare nelle variabili di sessione dei dati arbitrariamente grandi, che devono quindi essere memorizzati lato server.

Per memorizzare le variabili di sessione lato server generalmente viene utilizzata questa tecnica: la prima volta che un utente si visita l'applicazione Web, il server genera un ID casuale che identifica l'utente, memorizza questo ID nel browser dell'utente tramite un Cookie HTTP, e da quel momento in poi memorizza tutte le variabili di sessione di quell'utente su un database sul server, associandole all'ID di sessione dell'utente.

Questa tecnica ha il vantaggio di poter memorizzare grandi quantità di dati nelle variabili di sessione, e funziona piuttosto bene nel caso l'applicazione Web sia eseguita su un solo server. Ma cosa succede se invece devo eseguire l'applicazione su un numero elevato di server? In questo caso, uno stesso utente potrebbe effettuare più richieste che vengono indirizzate ognuna su server diversi. Siccome l'applicazione lato server deve sempre poter leggere e scrivere le variabili di sessione per ogni richiesta HTTP, se uno stesso utente esegue più richieste su server diversi, per ogni richiesta le variabili di sessione di quell'utente devono essere necessariamente trasferite dal server nel quale sono memorizzate a quello che sta servendo la richiesta. Questo fa si che una buona parte delle richieste HTTP per essere gestite richiedano l'intervento di 2 server invece che uno solo, in quanto per ogni richiesta va contattato anche il server nel quale sono memorizzate le variabili di sessione.

Come è facile intuire questo rappresenta un problema per la scabilità dell'applicazione Web, in quanto può crearsi un collo di bottiglia nel sistema di memorizzazione variabili di sessione: anche aumentando i server sul

quale viene eseguita l'applicazione Web, se non viene parallelamente aumentata anche il throughput del database delle variabili di sessione, il sistema complessivo non riesce a gestire un numero più alto di utenti.

Il Play framework risolve questo problema in modo piuttosto radicale: invece di memorizzare le variabili di sessione lato server come fanno la quasi totalità degli altri framework Web, Play memorizza le variabili di sessione esclusivamente lato client, cioè nei browser degli utenti, utilizzando i Cookie HTTP.

Questo approccio ha il grosso vantaggio di rendere le richieste HTTP effettuate dagli utenti completamente stateless, come dovrebbero in effetti essere secondo le specifiche del protocollo HTTP: infatti, memorizzando le variabili di sessione esclusivamente nei Cookie, ogni sessione può essere gestita interamente da un server Web qualsiasi senza che esso debba anche contattare un altro server per ottenere le variabili di sessione.

Con questo approccio quindi si guadagna in scalabilità, ma ha però anche degli svantaggi: siccome i Cookie HTTP hanno una dimensione massima di 4 kilobyte ed essi devono essere inviati ad ogni singola richiesta HTTP, le variabili di sessione in essi contenute devono essere di dimensioni molto piccole. Questo tuttavia non è un problema, in quanto nella maggior parte dei casi questi dati sono molto piccoli, e nei rari casi in cui sia necessario memorizzare dei dati più grandi basta inserire nella varabile di sessione un ID che in un database centrale lato server corrisponde al dato vero e proprio.

Inoltre è necessario risolvere un problema di sicurezza: l'utente non deve essere in grado di alterare i valori nelle variabili di sessione, in quanto per esempio in esse potrebbe essere memorizzato se l'utente ha il permesso o no di accedere ad un'area risevata. Questo problema viene completamente risolto da Play inserendo nei Cookie HTTP anche una firma digitale del contenuto delle variabili di sessione generata da una chiave di cui solo i server sono in possesso. In questo modo se il client altera i dati, i server se ne accorgono e rifiutano la richiesta.

## 4.2 I Websocket su Play: utilizzo esplicito degli Attori Akka

Dato che Play è basato su Akka, esso è particolarmente adatto per realizzare applicazioni che devono scambiarsi in modo bidirezionale dei messaggi asin-

croni, come quelli visti nel modello ad Attori. Come accennato nei capitoli precedenti, il protocollo migliore utilizzato in ambito Web per scambiarsi messaggi di questo tipo è quello dei Websocket. Grazie ad esso infatti possiamo dal server web inviare in qualunque momento messaggi al client.

In Play è possibile utilizzare i Websocket in modo molto semplice ma al tempo stesso estremamente potente, utilizzando gli Attori del sistema Akka sottostante: all'interno dell'applicazione Play, è possibile creare Attori che all'arrivo di certi messaggi da un qualsiasi altro Attore, inviano dei dati attraverso la connessione Websocket.

Se si vuole per esempio realizzare una applicazione Web che svolga lato server una computazione piuttosto complessa, si può progettare il tutto in questo modo: quando il browser effettua la richiesta al server, all'interno del sistema ad Attori Akka sul quale viene eseguita l'applicazione Play, verrà creato un Attore che eseguirà la computazione, eventualmente delegandola ad altri Attori, grazie allo scambio di messaggi. Quando gli Attori coinvolti avranno terminato il lavoro, il risultato della computazione verrà in qualche modo trasmesso con dei messaggi all'Attore supervisore che aveva dato inizio al lavoro. Esso infine, alla ricezione di questo risultato, potrà inviarlo tramite Websocket al browser dell'utente.

In questo modo, il trasferimento del risultato del lavoro viene lanciato senza nessun intervento da parte del client, ma solamente in seguito alla ricezione di un messaggio da parte di un certo Attore dell'applicazione Play.

Come esempio pratico di codice, vediamo un'applicazione che svolge un lavoro simile a quello sopra descritto: vogliamo realizzare una pagina Web in cui viene visualizzato un orologio, nel quale le cifre dell'ora da visualizzare non vengono aggiornate dal client con il passare dei secondi, ma devono essere inviate di volta in volta dal server. Ogni secondo il server dovrà quindi inviare al client l'ora attuale tramite un Websocket, e di conseguenza del codice lato client aggiornerà l'ora visualizzata nell'orologio.

Quando l'utente dal browser visiterà la pagina Web in cui deve essere visualizzato questo orologio ed instaurerà la connessione Websocket, nell'applicazione Play sul server dovrà essere creato questo Attore Akka che si occuperà di inviare gli aggiornamenti dell'ora tramite il Websocket:

```
public static class Clock extends UntypedActor {

  static ActorRef instance =
```

```
    Akka.system().actorOf(new Props(Clock.class));

// Send a TICK message every 100 millis
static {
    Akka.system().scheduler().schedule(
        Duration.Zero(),
        Duration.create(100, MILLISECONDS),
        instance, "TICK"
    );
}

List<WebSocket> sockets = new ArrayList<WebSocket>();
SimpleDateFormat dateFormat = new SimpleDateFormat("HH mm ss");

public void onReceive(Object message) {

    // Handle connections
    if(message instanceof WebSocket) {
        final WebSocket socket = (WebSocket)message;
        if(sockets.contains(socket)) {
            // Brower is disconnected
            sockets.remove(socket);
            Logger.info("Browser disconnected ("
                + sockets.size() + " browsers currently connected)");
        } else {
            // Register disconnected callback
            socket.onDisconnected(new Callback0() {
                public void invoke() {
                    getContext().self().tell(socket);
                }
            });

            // New browser connected
            sockets.add(socket);
            Logger.info("New browser connected ("
                + sockets.size() + " browsers currently connected)");
        }
```

```
        }

        // Tick, send time to all connected browsers
        if("TICK".equals(message)) {
            // Send the current time to all WebSocket sockets
            for(WebSocket socket: sockets) {
                socket.sendMessage(dateFormat.format(new Date()));
            }
        }
    }
}
```

Questo Attore riceverà dallo scheduler di Akka un messaggio ogni secondo contenente la stringa "TICK", in risposta al quale invierà la data e l'ora tramite Websocket a tutti i browser connessi. Per tenere traccia dei WebSocket attualmente connessi, cioè dei browser ai quali vanno inviati i messaggi, viene utilizzata la ArrayList sockets, presente tra i campi della classe di questo Attore. Quando un browser si connette, viene inviato a questo Attore il WebSocket che verrà quindi aggiunto alla lista. Quando un browser si disconnette esplicitamente oppure va in timeout la connessione TCP sottostante (per esempio perchè il browser è andato in crash), viene chiamato un metodo di callback che farà in modo rimuovere il WebSocket corrispondente dalla lista.

Questo semplice esempio può essere ovviamente esteso a software molto più complessi, nei quali vari Attori sul server effettuano lavori molto più complessi, ed alla fine l'Attore che si ritrova dentro di se il risultato completo del lavoro, lo invia al client tramite un Websocket.

Nella prossima sezione riprenderemo questa applicazione di esempio che mostra un orologio, vedendone una versione leggermente modificata, che utilizza i socket Comet invece che i WebSocket. I socket Comet [6] sono una sorta di protocollo che offre delle funzionalità quasi equivalenti a quello dei WebSocket, ma sono implementati utilizzando semplicemente il protocollo HTTP standard, grazie a delle tecniche di long-polling.

Dato che il protocollo HTTP non è stato progettato per il trasferimento bidirezionale dei dati come avviene nei WebSocket, il sistema dei Comet è considerato una sorta di hack ed è preferibile se possibile non usarlo, tuttavia esso ha il grosso vantaggio di essere compatibile con quasi la totalità dei browser e webserver in commercio, anche quelli meno recenti. La maggior

parte delle applicazioni Web attuali che richiedono delle funzionalità simili a quelle dei WebSocket, generalmente vengono realizzate con un sistema ibrido: nel caso che il browser dell'utente supporti i WebSocket, vengono utilizzati i WebSocket sfruttandone pienamente tutti i vantaggi, altrimenti come fallback vengono utilizzati i socket Comet.

Varie applicazioni Web molto famose, come quelle di Facebook e di Google, utilizzano questo sistema ibrido appena descritto. Molti servizi di hosting ed alcuni servizi di Cloud Computing come Google App Engine ed Heroku, ancora non hanno purtroppo introdotto il supporto ai WebSocket. Nella applicazione di esempio che vedremo nella prossima sezione, dato che dovremo pubblicarla sulla piattaforma Heroku, utilizzeremo i socket Comet invece che i WebSocket. Questo non è tuttavia un problema, in quanto il Play framework permette di usare i socket Comet in modo molto semplice ed elegante, offrendo allo sviluppatore delle API pressochè identiche a quelle dei WebSocket [7] .

## 4.3 Deploy di una applicazione Play sulla piattaforma Cloud Heroku

### 4.3.1 L'entrata in produzione di un'applicazione Play

Finora abbiamo visto come progettare e sviluppare un'applicazione Web basata sul Play framework. Ora vediamo invece la fase finale della realizzazione dell'applicazione, cioè il renderla disponibile al pubblico.

Ovviamente, come tutte le Web application, è necessario far si che l'applicazione Play possa essere eseguita su almeno un server raggiungibile via rete dagli utenti. Come già accennato però, al contrario della maggior parte delle altre piattaforme di sviluppo Web come PHP, il Play framework integra dentro di se anche un vero e proprio server HTTP, JBoss Netty per la precisione, opportunamente adattato per essere perfettamente integrato con il resto del framework. Se non si hanno esigenze particolari, è quindi possibile eseguire l'applicazione Play su una JVM su un proprio server, e far connettere direttamente gli utenti al server HTTP Netty integrato in Play. In certi casi tuttavia può essere necessario inserire un reverse proxy tra l'applicazione Play e gli utenti, per ottenere alcune funzionalità come la spartizione del traffico su più webserver.

Aumentando la complessità ed il numero di utenti dell'applicazione Web, distribuire il traffico su più server diventa quasi indispensabile, e quindi la gestione della propria infrastruttura server può diventare anche molto complessa, sopratutto se vanno effettuate operazioni relativamente di basso livello come la manutenzione e l'aggiornamento dei sistemi operativi dei server.

Per risolvere questo problema sono nate le varie soluzioni di Cloud Computing, tra le quali per esempio Heroku, che ora vedremo brevemente.

### 4.3.2 Heroku: che cos'è e come funziona

Heroku è un servizio di Cloud Computing che offre una piattaforma per eseguire e rendere disponibili al pubblico delle applicazioni Web realizzate su certi framework e linguaggi da essa supportati.

Heroku rientra nella categoria delle PaaS, cioè Platform as a Service, in quanto offre una piattaforma sulla quale eseguire le proprie applicazioni senza preoccuparsi di dettagli implementativi come il gestire gli aggiornamenti ed i guasti dei server sui quali l'applicazione verrà eseguita. Utilizzare una PaaS è quindi molto semplice, in quanto basta inserire il codice della propria applicazione nella piattaforma, generalmente tramite una apposita console di sviluppo, ed essa si occuperà di tutto il resto delle operazioni. Tra le varie PaaS disponibili sul mercato, analizziamo Heroku perchè è stata progettata per supportare pienamente le applicazioni realizzate con il Play framework.

Utilizzando una piattaforma come Heroku, la propria applicazione diventa già dalla partenza scalabilie ad un numero teoricamente illimitato di utenti, in quanto la piattaforma si occupa di allocare ulteriori server alla propria applicazione. Questa operazione di distribuzione su più server, nel caso delle applicazioni basate sul Play framework, è completamente trasparente sia agli utenti, sia all'applicazione stessa, grazie alle funzionalità precedentemente spiegate del framework Play e di Akka. La piattaforma Heroku si occuperà anche di gestire, in modo completamente trasparente, dei reverse proxy che distribuiscono le richieste provenienti dai client su più server, e svolgono funzioni di caching.

Heroku offre anche dei servizi di database e storage di vario genere, che possono essere utilizzati dalle proprie applicazioni, sempre senza dover effettuare alcuna operazione di gestione dei server.

### 4.3.3 Pubblicazione di una Web application Play su Heroku

Dato che il Play framework si presta particolarmente per essere utilizzato nelle piattaforme Cloud per i motivi precedentemente descritti, vari servizi di PaaS permettono di utilizzare le applicazioni Play nelle loro piattaforme. Fra di questi il servizio più famoso ed utilizzato è sicuramente Heroku, il quale offre tra l'altro dei tool che permettono di pubblicare delle applicazioni Play su Heroku in modo molto semplice. [11]

Per pubblicare delle applicazioni su Heroku, occorre registrarsi al servizio e scaricare l'Heroku Toolbelt, dei tools da installare nel computer dello sviluppatore (sono disponibili per Linux, Mac e Windows) che permettono di eseguire varie operazioni in modo semplice dal terminale. Heroku offre gratuitamente la possibilità di pubblicare un'applicazione che utilizzi un solo Dyno, cioè una sorta di CPU virtuale, un modo ideato dagli sviluppatori di Heroku per misurare la quantità di risorse dei server utilizzate da una certa applicazione. Dato che la nostra applicazione di esempio avrà pochissimo traffico, può essere tranquillamente pubblicata gratuitamente su Heroku.

Come applicazione Play di esempio da pubblicare su Heroku, riprendiamo quella vista nella sezione precedente sui WebSocket: una applicazione che mostra nel browser dell'utente un orologio, la cui ora visualizzata viene aggiornata tramite dei messaggi provenienti dal server. In questo caso, come già spiegato nella sezione precedente, utilizzeremo i socket Comet invece che i WebSocket, in quanto Heroku supporta per il momento solo i Comet.

Prima di descrivere la procedura di deploying su Heroku, vediamo il codice dei file sorgenti più importanti della nostra applicazione Play:

Questo è il codice del file app/controllers/Application.java, che contiene la logica del controller:

```
package controllers;

import play.*;
import play.mvc.*;
import play.libs.*;
import play.libs.F.*;
import akka.util.*;
import akka.actor.*;
import java.util.*;
```

```
import java.text.*;
import static java.util.concurrent.TimeUnit.*;
import views.html.*;

public class Application extends Controller {

    final static ActorRef clock = Clock.instance;

    public static Result index() {
        return ok(index.render());
    }

    public static Result liveClock() {
        return ok(new Comet("parent.clockChanged") {
            public void onConnected() {
               clock.tell(this);
            }
        });
    }

    public static class Clock extends UntypedActor {

        static ActorRef instance =
            Akka.system().actorOf(new Props(Clock.class));

        // Send a TICK message every 100 millis
        static {
            Akka.system().scheduler().schedule(
                Duration.Zero(),
                Duration.create(100, MILLISECONDS),
                instance, "TICK"
            );
        }

        List<Comet> sockets = new ArrayList<Comet>();
        SimpleDateFormat dateFormat =
               new SimpleDateFormat("HH mm ss");
```

```
        public void onReceive(Object message) {

            // Handle connections
            if(message instanceof Comet) {
                final Comet cometSocket = (Comet)message;

                if(sockets.contains(cometSocket)) {

                    // Brower is disconnected
                    sockets.remove(cometSocket);

                } else {

                    // Register disconnected callback
                    cometSocket.onDisconnected(new Callback0() {
                        public void invoke() {
                            getContext().self().tell(cometSocket);
                        }
                    });

                    // New browser connected
                    sockets.add(cometSocket);
                }
            }

            // Tick, send time to all connected browsers
            if("TICK".equals(message)) {

                // Send the current time to all comet sockets
                for(Comet cometSocket: sockets) {
                    cometSocket.sendMessage(
                        dateFormat.format(new Date()));
                }
            }
        }
    }
```

```
}
```

Come si può notare, esso è molto simile al codice di esempio visto in precedenza riguardo ai WebSocket. L'unica differenza tra i due è che quello precedente utilizzava i WebSocket, questo invece utilizza i socket Comet.

Il codice HTML5 della pagina Web dell'applicazione, come descritto in precedenza, viene creato dal sistema di templating del Play framework. Dato che nella nostra applicazione di esempio il codice HTML di output della pagina è sempre lo stesso, riporto per brevità solo il codice HTML prodotto in output dal sistema di templating ed inviato al browser:

```
<!DOCTYPE html>

<html>
  <head>
    <title>Comet clock</title>
    <link rel="stylesheet" href="/assets/stylesheets/main.css">
    <script src="/assets/javascripts/jquery-1.7.1.min.js"
                              type="text/javascript"></script>
    </head>
    <body>

    <h1>Comet clock</h1>

    <h1 id="clock"></h1>
    <p>
        L'ora corrente viene inviata dai server Heroku
        tramite un socket Comet
    </p>

    <script type="text/javascript" charset="utf-8">
      // Called for each Comet message
      var clockChanged = function(time) {
        $('#clock').html(time.replace(/(\d)/g, '<span>$1</span>'))
      }
    </script>

    <iframe id="comet" src="/clock?1805389028007559372"></iframe>
```

```
    </body>
</html>
```

L'elemento iframe fa parte del sistema Comet ed è gestito completamente dalle librerie del framework Play, quindi lo sviluppatore dell'applicazione Web non si deve occupare di esso. Ciò che invece ci interessa come sviluppatori della presente Web application, è il codice JavaScript presente subito sopra il tag iframe. Esso utilizza la libreria jQuery per ricevere i dati dell'ora corrente dal socket Comet, ed inserirli nel DOM della pagina HTML, facendoli visualizzare quindi nel browser dell'utente.

Vediamo ora invece la procedura vera e propria di pubblicazione dell'applicazione su Heroku: iniziamo aprendo un terminale e posizionandoci sulla directory root dell'applicazione Play. Successivamente creiamo un repository git locale nella directory corrente ed aggiungiamogli tutto il codice dell'applicazione Play eseguendo un commit:

```
$ git init
$ git add .
$ git commit -m "init"
```

Ora utilizziamo l'Heroku Toolbelt per creare effettuare il login con il nostro account Heroku, e per creare un applicazione Heroku dal codice nella directory corrente:

```
$ heroku login
$ heroku create
```

L'Heroku Toolbelt stamperà in standard output l'URL pubblico a cui sarà raggiungibile l'applicazione Heroku. L'URL fornito in questo momento è di questo tipo:

```
http://rocky-brushlands-6234.herokuapp.com/
```

ed è ovviamente solo un URL temporaneo adatto per il testing della nostra applicazione. Successivamente è possibile associargli il proprio dominio, ed ottenere quindi l'URL desiderato.

Ora per pubblicare l'applicazione su Heroku, dobbiamo eseguire un push sul repository git remoto di Heroku che ha configurato l'Heroku Toolbelt nel passo precendente:

```
$ git push heroku master
```

La pubblicazione di una applicazione su Heroku viene quindi vista come l'invio di una certa branch della propria applicazione al repository git nei server di Heroku. Questa metodologia di deploying delle Web application è particolarmente vantaggiosa, in quanto permette agli sviluppatori di gestire il lifecycle della propria applicazione interamente con git, il sistema MVCC attualmente considerato tra i più avanzati attualmente disponibili ed utilizzato in una grandissima quantità di progetti open-source e non.

Una volta terminata l'operazione, l'applicazione sarà resa disponibile sulla piattaforma Heroku all'URL precedentemente indicato. Questo è il risultato che si ottiene accedendo all'URL della nostra Web application di esempio da un browser:


Da notare che l'orario indicato nella pagina Web dell'applcazione, è un'ora indietro rispetto all'orario del computer su cui è stato eseguito il browser. Questo dimostra che l'orario è stato inviato dai server Heroku, i quali utilizzano l'orario UTC, mentre sul sistema Linux del computer su cui ho effettuato lo screenshot è impostata l'ora UTC+1 (ora solare italiana).

# Capitolo 5

# Conclusioni

In questa Tesi abbiamo affrontato il problema di come realizzare applicazioni Web moderne e scalabili, in modo semplice ed orientato all'uso del Cloud.

Per raggiungere questo scopo, è stato studiato il modello ad Attori, che è risultato piuttosto utile in vari ambiti della programmazione Web. Si è visto che grazie al modello ad Attori, è possibile progettare in modo semplice del software con un elevato grado di concorrenza, che altrimenti sarebbe risultato più difficoltoso sia in fase di progettazione che di realizzazione.

Questo è stato utile sia per ciò che riguarda il lato client delle applicazioni Web, tramite i Web Worker che abbiamo visto brevemente, sia per quanto riguarda il lato server, per il quale abbiamo analizzato il Play framework. Grazie ad esso abbiamo visto che è possibile realizzare in modo molto semplice applicazioni Web moderne anche piuttosto complesse, grazie alle sue varie funzionalità fornite anche dal framework Akka sottostante. Play inoltre consente di realizzare applicazioni Web estremamente scalabili e predisposte già dalla partenza ad essere eseguite su piattaforme di Cloud Computing PaaS innovative come Heroku.

In conclusione abbiamo quindi visto un insieme di concetti e strumenti idonei per sviluppare applicazioni Web moderne, scalabili ed orientate al Cloud, come sono sempre di più richieste attualmente dal mondo dell'industria.

# Ringraziamenti

Desidero innanzitutto ringraziare il Prof. Alessandro Ricci per avermi dato preziosi consigli ed avermi seguito nella stesura di questa tesi.

Ringrazio inoltre i miei amici, la mia famiglia, e tutti coloro che mi sono stati vicini durante questi anni di studio.

# Bibliografia


[1] Gul Agha.
Concurrent object-oriented programming.
Communications of the ACM, 1990.

[2] Rajesh K. Karmani, Amin Shali, Gul Agha.
Actors Frameworks for the JVM Platform: A Comparative Analysis.
ACM, Association for Computing Machinery, 2009.

[3] Multi-Processing Modules (MPMs), documentazione dell'Apache HTTP Server, 2012.
`http://httpd.apache.org/docs/2.4/mpm.html`.

[4] Sito ufficiale di nginx, 2012.
`http://wiki.nginx.org/Main`.

[5] The C10K problem, 2012.
`http://www.kegel.com/c10k.html`.

[6] Sito ufficiale di Akka, 2012.
`http://akka.io/`.

[7] Actors, documentazione di Akka (Java), 2012.
`http://doc.akka.io/docs/akka/2.0.4/java/untyped-actors.html`.

[8] Remoting, documentazione di Akka (Java), 2012.
`http://doc.akka.io/docs/akka/2.0.4/java/remoting.html`.

[9] Introduzione al Play framework 2.0, 2012.
`http://www.playframework.org/documentation/2.0.4/Philosophy`.

[10] Articolo sul modello Comet di Wikipedia, 2012.
`http://en.wikipedia.org/wiki/Comet_(programming)`.

[11] Comet sockets, documentazione del Play framework, 2012.
`http://www.playframework.org/documentation/2.0.4/JavaComet`.

[12] Sito ufficiale di Ebean, 2012.
`http://www.avaje.org/ebean/introduction.html`.

[13] Ebean ORM, documentazione del Play framework, 2012.
`http://www.playframework.org/documentation/2.0.4/JavaEbean`.

[14] Sito ufficiale di Heroku, 2012.
`http://www.heroku.com/`.

[15] Deploying to Heroku, documentazione del Play Framework, 2012.
`http://www.playframework.org/documentation/2.0.4/ProductionHeroku`.