ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

REACTIVE PROGRAMMING: UN CASO DI STUDIO

Elaborato in
TECNOLOGIE WEB

Relatore
Prof. ALESSANDRO RICCI

Co-relatore
FRANCESCO FULLONE

Presentata da
MARTINA GIOVANELLI

Terza Sessione di Laurea
Anno Accademico 2014 – 2015

PAROLE CHIAVE

Reactive programming

Asynchronous stream

ReactiveX

RxJS

RxPHP

Alla mia famiglia

Indice

Introduzione **ix**

1 Reactive Programmig **1**
1.1 La programmazione reattiva 1
1.1.1 La propagazione del cambiamento 1
1.1.2 Flussi asincroni di dati e eventi 3
1.2 Scenari di utilizzo 4
1.3 Caratteristiche dei linguaggi di programmazione reattivi 5
1.3.1 Astrazioni base 5
1.3.2 Modelli di valutazione 6
1.3.3 Glitch 7
1.3.4 Operazioni di sollevamento 9
1.3.5 Multidirezionalità 10
1.3.6 Supporto per il distribuito 10
1.4 Reactive manifesto 10
1.5 Classificazione dei linguaggi reattivi 14

2 Esempi di tecnologie: RxJS e RxPHP **17**
2.1 Reactive Extension 17
2.2 RxJS 22
2.3 RxPHP 39

3 Caso di studio: il progetto AvvocaTimer **43**
3.1 Introduzione 43
3.2 Analisi dei requisiti 45
3.3 Progettazione 47
3.3.1 Parte front-end 47
3.3.2 Parte back-end 52
3.4 Implementazione 53
3.4.1 Parte front-end 53
3.4.2 Parte back-end 62
3.5 Collaudo 63

4 La programmazione reattiva a confronto con gli altri approcci per la programmazione asincrona 67
4.1 La gestione di codice asincrono 67
4.2 Codice asincrono in Javascript 68
4.3 Codice asincrono con RxJS 71

Conclusioni 77

Ringraziamenti 79

Bibliografia 81

Introduzione

L'obiettivo di questa tesi è analizzare e testare la programmazione reattiva, paradigma di programmazione particolarmente adatto per lo sviluppo di applicazioni altamente interattive, che devono reagire prontamente agli eventi che avvengono nell'ambiente circostante e elaborare dati provenienti da diverse sorgenti. La progettazione di sistemi reattivi implica necessariamente l'utilizzo di codice asincrono e la programmazione reattiva (RP) offre al programmatore semplici meccanismi per gestirlo.

In questa tesi, la programmazione reattiva è stata utilizzata e valutata mediante la realizzazione di un progetto real-world chiamato `AvvocaTimer`, progettato e sviluppato all'interno dell'azienda Ideato srl. `AvvocaTimer` è un sistema di time tracking per studi legali che permette di monitorare il lavoro dei vari avvocati, la gestione dell'archivio di casi e clienti, la coordinazione di più uffici, la visualizzazione di informazioni di sintesi circa il lavoro effettuato e non solo. Il sistema consiste principalmente di una web application e una applicazione mobile.

Sono state sviluppate due versioni per la web application: nella prima release il codice asincrono viene gestito con i meccanismi attualmente utilizzati, nella seconda, invece, è stato usato il nuovo approccio reattivo.
In questa tesi verrà affrontata la progettazione, implementazione e collaudo della web application e di una piccola parte di back-end con la programmazione reattiva. Successivamente, le due versioni (reattiva e non) verranno confrontate e, analizzati vantaggi e/o svantaggi derivanti dall'utilizzo del nuovo paradigma.

Si è voluto mostrare come sia possibile rendere le applicazioni reali molto più robuste integrandole con la programmazione reattiva e come questa possa essere utilizzata per gestire gli aspetti asincroni del front-end e del back-end delle moderne applicazioni.

La tesi è suddivisa in quattro capitoli.

Nel primo vengono analizzati i linguaggi di programmazione reattivi sulla

base di determinati criteri e, rappresentata la loro relazione con il manifesto reattivo.

Nel secondo capitolo verranno trattate le tecnologie utilizzate per la rivisitazione in chiave reattiva della web application e parte del back-end cioè della libreria Reactive Extension con particolare riferimento alle estensioni reattive previste da questa per Javascript e PHP.

Il terzo capitolo include l'analisi dei requisiti, la progettazione e implementazione delle parti precedentemente individuate con il nuovo paradigma.

Il quarto capitolo tratta brevemente degli approcci correntemente utilizzati per la gestione di codice asincrono, quelli utilizzati in Javascrit e dei loro svantaggi. Infine, vengono confrontate le due versioni dell'applicazione e mostrati i vantaggi ottenuti integrando la prima release con la programmazione reattiva.

Segue una breve conclusione.

Capitolo 1

Reactive Programmig

1.1 La programmazione reattiva

Nel web è possibile trovare molte definizioni diverse per programmazione reattiva, vediamo alcune.
Per Microsoft:
Reactive Extensions is for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators.
Per Wikipedia:
Reactive Programming is a programming paradigm oriented around data flows and the propagation of change.
Per Reactive Manifesto:
Reactive Systems are Responsive, Elastic, Resilient, Message Driven.

La definizione fornita da Microsoft può dar luogo a confusione, Wikepedia è molto generica e teorica mentre, per il manifesto reattivo sembrerebbe una politica aziendale. Da tali definizioni, però, è possibile ricavare i concetti sui quali la programmazione reattiva (RP) si basa:

- incentrata sul concetto di flussi di dati e la propagazione del cambiamento.
- Permette di gestire facilmente flussi asincroni di dati ed eventi (asynchronous streams).
- Fortemente correlata alla programmazione ad eventi.

1.1.1 La propagazione del cambiamento

La programmazione reattiva è un paradigma di programmazione costruito sulla nozione del cambiamento dei valori nel tempo e la propagazione dei cam-

biamenti. Essa facilita lo sviluppo di applicazioni event-driven permettendo allo sviluppatore di esprimere programmi in termini di che cosa devono fare, consentendo al linguaggio di gestire automaticamente quando farlo. In questo paradigma, i cambiamenti di stato sono efficientemente e automaticamente propagati attraverso la rete delle dipendenze computazionali dal sottostante modello di esecuzione. Di seguito proviamo a spiegare la propagazione del cambiamento attraverso un semplice esempio di somma di due variabili.

$$\text{var1} = 1$$
$$\text{var2} = 2$$
$$\text{var3} = \text{var1}+\text{var2}$$

Convenzionalmente, in un linguaggio imperativo sequenziale, se il programmatore non assegna esplicitamente un nuovo valore a var3 il suo valore sarà sempre 3 cioè la somma dei valori iniziali delle variabili var1 e var2, anche quando, successivamente, verranno assegnati nuovi valori a queste. Nella programmazione reattiva il valore della variabile var3 è sempre mantenuto aggiornato cioè, viene automaticamente ricalcolato dopo ogni cambiamento di var1 o di var2. Questa è una nozione chiave della programmazione reattiva. I valori cambiano nel tempo e quando ciò avviene ogni espressione che dipende da questi viene automaticamente ricalcolata. Nella terminologia della programmazione reattiva, la variabile var3 dipende dalle variabili var1 e var2. E' possibile rappresentare questa dipendenza attraverso un grafico detto grafico delle dipendenze (figura 1.1).

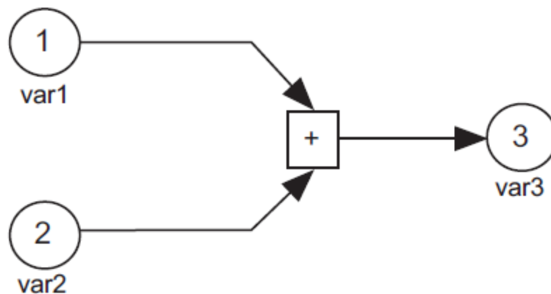

Figura 1.1: Esempio grafico delle dipendenze nella programmazione reattiva

Questo concetto di gestione automatica delle dipendenze avviene anche all'interno di un qualsiasi foglio di calcolo elettronico; se vi riportiamo questa formula, ad ogni cambiamento delle celle che contengono i primi due valori, cambierà automaticamente anche il valore della cella che contiene la loro somma. La programmazione reattiva può essere considerata un foglio elettronico incorporato in un linguaggio di programmazione.

1.1.2 Flussi asincroni di dati e eventi

La programmazione reattiva permette di gestire facilmente flussi asincroni di dati ed eventi. Si possono creare flussi di dati da qualsiasi cosa come, dal click del mouse, da variabili, dall'input di un utente, dai feed di twitter ecc... Vengono, inoltre, fornite diverse funzioni per combinare, creare o interrogare questi flussi. Esempi di operazioni che posso fare con gli stream: passarlo in input o unirlo ad un altro, ottenerne uno nuovo con solo gli eventi che rispettano un determinato predicato di selezione, rappresentare i suoi elementi in un modo diverso in un altro stream ecc.. Essi rappresentano un altro aspetto chiave della programmazione reattiva. Consideriamo un esempio concreto di flusso di eventi e,in particolare, il click su un bottone.

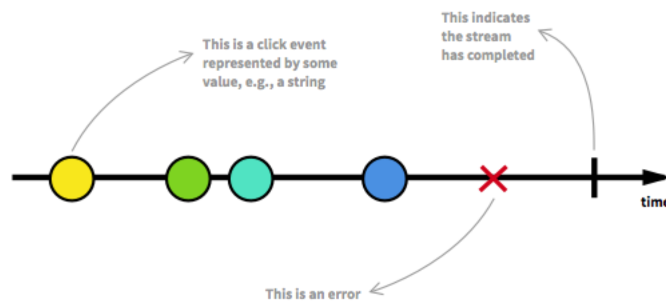

Figura 1.2: Observable generato dagli eventi click di un bottone

Lo stream è un seguenza di eventi ordinati nel tempo che può o non terminare. Anche una lista o una collezione di dati può essere considerata come un stream: un stream finito e deterministico. Esso può emettere tre differenti cose: un nuovo valore, un errore o un segnale che indica il fatto che è completo cioè che dallo stream non verranno emessi più valori. Nel nostro esempio il segnale 'completo' potrà essere emesso quando la finestra che contiene il bottone del quale stiamo gestendo l'evento click viene chiusa. Questi eventi vengono catturati in modo asincrono definendo delle funzioni che verranno eseguite rispettivamente quando un nuovo valore viene emesso, quando avviene un errore o quando lo stream termina. Viene detto sottoscrittore o osservatore colui che rimane in ascolto di nuovi valori mentre, osservabile lo stream di dati o eventi (pattern Observer). Non è possibile modificare gli stream poiché sono immutabili: ogni operazione,infatti, ne produce uno nuovo.

1.2 Scenari di utilizzo

Figura 1.3: Scenari di utilizzo

La programmazione reattiva negli ultimi anni è diventata popolare come paradigma di programmazione il cui obiettivo è quello di semplificare l'implementazione di applicazioni event-driven e fornire un semplice meccanismo per lavorare con codice asincrono. Il comportamento delle moderne applicazioni è guidato da ogni sorta di evento generato all'interno del contesto dell'applicazione (è terminata la scrittura su un file, è giunta la risposta da un server web...) o dall'ambiente esterno (cambiano la coordinate del GPS, messaggi o like di un social network...) figura (1.3). Queste applicazione hanno la necessità di interagire continuamente con l'ambiente che le circondano e reagire prontamente agli eventi che arrivato eseguendo determinati compiti come, ad esempio, aggiornare lo stato o visualizzare dati. Generalmente, la parte più interattiva di una applicazione è la GUI, la quale deve reagire e coordinare molti eventi (il click del mouse, la pressione di un pulsante della tastiera, il multitouch ecc...)

Reagire ad eventi nel classico modo imperativo porta alla scrittura di codice ingombrante, difficile da comprendere e, talvolta, è molto semplice commettere errori poiché la coordinazione fra evento e i cambiamenti ai dati è responsabilità del programmatore il quale deve fare tutto questo manualmente in porzioni di codice separate che possono manipolare gli stessi dati e che potrebbero essere eseguite in qualsiasi ordine nel tempo. Usando le tradizionali soluzioni le applicazioni interattive sono costruite sulla nozione di callbacks asincrone ma la loro gestione non è affatto facile poiché si hanno frammenti di codice isolato del quale non si conosce l'ordine di esecuzione (il flusso di controllo del

programma 'salta' fra i vari event handlers, non ha un ordine specificato dal programmatore)

La programmazione reattiva prevede astrazioni per rappresentare gli eventi e il cambiamento dei valori nel tempo, liberando il programmatore dalla gestione delle dipendenza fra questi. Essa non è solo un paradigma di programmazione client-side ma utilissimo anche in contesti server-side.

Viene utilizzata in grandi applicazioni come Cortana, Netflix ecc...

1.3 Caratteristiche dei linguaggi di programmazione reattivi

Passiamo ora ad una breve descrizione delle principali caratteristiche dei linguaggi di programmazione reattivi e, in particolare, verranno analizzate:

- le astrazioni base
- il modello di valutazione (evaluation model)
- le operazioni di sollevamento (lifting operations)
- linguaggi multi-direzionali (multidirectionality)
- problemi tecnici come i glitch (glitch avoidance)
- il supporto per il distribuito (support for distribution)

1.3.1 Astrazioni base

COMPORTAMENTI (BEHAVIORS) Comportamento è il termine utilizzato per esprimere il cambiamento dei valori nel tempo. Il comportamento cambia continuamente ed, un suo esempio base, potrebbe essere il tempo stesso. In un linguaggio di programmazione reattivo il comportamento può essere espresso come semplice funzione del tempo. Altri esempi di comportamento sono:

- la finestra di un browser
- la posizione del cursore
- ...

EVENTI(EVENTS) Quando si parla di eventi ci si riferisce ad uno stream, potenzialmente infinito, di cambiamenti ai valori. A differenza del comportamento che continuamente cambia nel tempo, gli eventi avvengono in istanti ben precisi. Esempi di eventi sono:

- pressione di un bottone della tastiera
- click del mouse
- cambiamento della posizione
- ....

1.3.2 Modelli di valutazione

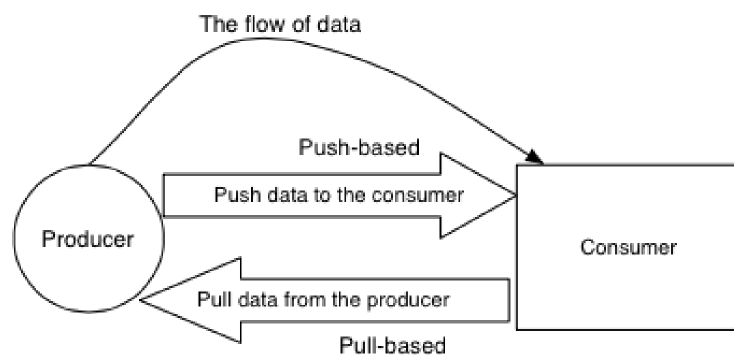

Figura 1.4: modelli di valutazione pull-based and push-based a confronto

Il modello di valutazione di un linguaggio di programmazione reattivo riguarda il modo con cui i cambiati sono propagati attraverso il grafo delle dipendenze. Dal punto di vista del programmatore la propagazione dei cambiamenti avviene automaticamente (un cambiamento ad un valore dovrebbe essere automaticamente propagato ad ogni sua dipendenza computazionale). Quando avviene un evento da una sorgente di eventi, le computazioni dipendenti devono essere notificate del cambiamento e dovono essere nuovamente valutate. A livello di linguaggio occorre stabilire chi da origine alla propagazione dei cambiamenti: è la sorgente che dovrebbe spingere "push" nuovi dati alle sue dipendenze(consumatori) o sono le dipendenze che dovrebbero tirare "pull" dati dalla sorgente di eventi(produttore) (figura 1.4). Nella letteratura della programmazione reattiva esistono due modelli di valutazione:

- PULL-BASED
  Nel modello pull-based le computazioni che necessitano di valori hanno

bisogno di 'tirarli' dalla sorgente. La propagazione è guidata dalla domanda di un nuovo valore (deman driven). Questo modello è abbastanza flessibile poiché le computazioni che richiedono dei valori posso "tirarli" quando necessitano e non a ogni cambiamento di questi. Si potrebbero, però, verificare latenze significative fra il momento in cui si verifica un evento e il momento in cui avviene la relativa reazione.

- PUSH-BASED
  Nel modello push-based, quando la sorgente ha nuovi dati, questi vengono 'spinti ' ad ogni loro dipendenza computazionale. La propagazione è guidata dalla disponibilità di nuovi dati (data driven) piuttosto che dalla domanda. I linguaggi che implementano un modello push-based hanno bisogno di una efficiente soluzione al problema di computazioni superflue poiché ad ogni cambiamento della sorgente ha luogo una nuova rivalutazione di tutte le dipendenze.

Esistono linguaggi di programmazione reattiva che adottano entrambi i modelli e quindi possono usufruire dei vantaggi offerti dal modello push-based(efficienza e bassa latenza) e dal modello pull-based (la flessibilità di richiamare valori all'occorrenza).

1.3.3 Glitch

Una proprietà che occorre considerare in un linguaggio reattivo è la sua capacità di evitare glitch, in questo contesto, vengono chiamati così gli aggiornamenti inconsistenti di espressioni che possono avvenire durante la propagazione dei cambiamenti. Questo fenomeno si potrebbe verificare nel caso in cui, un calcolo viene eseguito prima della valutazione di tutte le sue dipendenze cioè vengono combinati valori aggiornati e non. Ciò può succedere solo in linguaggi che adottano un modello di valutazione push-based. Consideriamo un esempio nella programmazione reattiva:
(1) var1 = 1
(2) var2 = var1 * 1
(3) var3 = var1 + var2

In questo esempio, il valore della variabile var2 sarà sempre uguale al valore di var1 e, il valore di var3 sarà sempre uguale al doppio di var1. Se il valore di var1 cambia in 2 ci si aspetta che var2 assuma il valore di 2 e var3 di 4. In alcune implementazioni reattive potrebbe accadere che la valutazione dell'espressione (3) avvenga prima della (2) e che var3 assuma momentaneamente il valore incorretto di 3. Solo dopo la rivalutazione dell'espressione (2) verrà

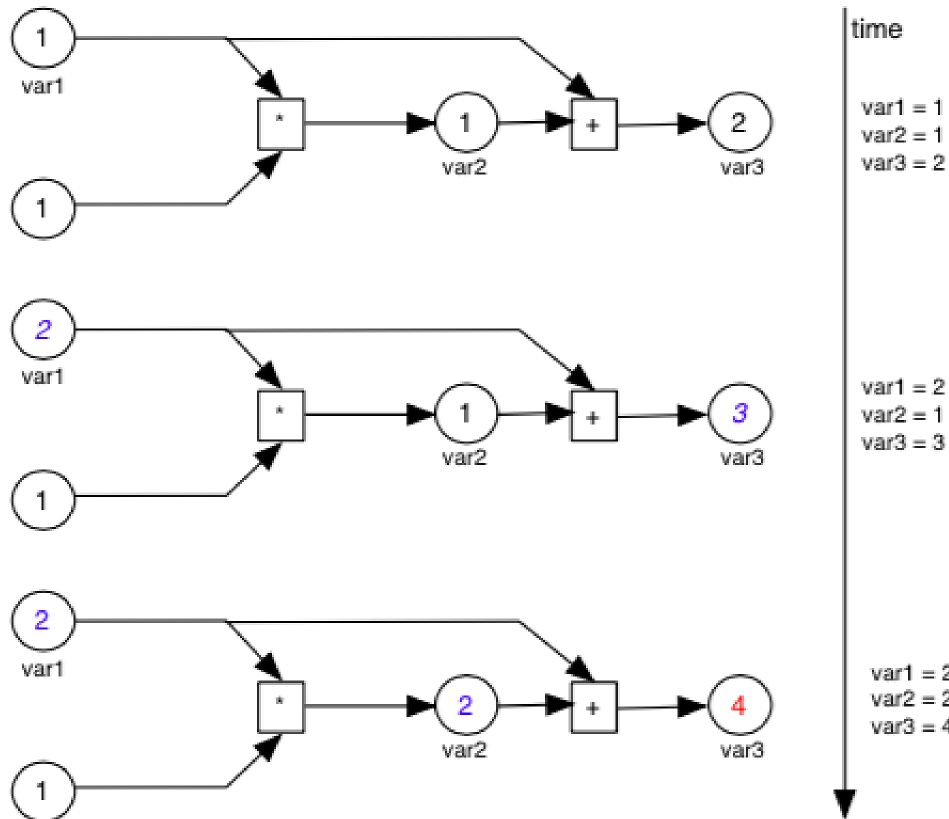

Figura 1.5: Glitch

nuovamente ricalcolata la (3) portando var3 ad un valore consistente. Questo comportamento viene descritto nella figura sotto (1.5).

Nella letteratura della programmazione reattiva questi aggiornamenti inconsistenti e superflui vengono chiamati glitch. Molti linguaggi di programmazione reattivi per eliminare tale fenomeno organizzano le espressioni in un grafo topologico ordinato: questo garantisce che ogni espressione venga valutata dopo che tutte le sue dipendenza sono già stata ricalcolate. Inoltre, una efficiente implementazione reattiva dovrebbe evitare rivalutazioni non necessarie di valori che non cambiano; riprendendo l'esempio sopra, se var1 inizialmente uguale ad 1 viene aggiornata allo stesso valore non dovrebbe avvenire la rivalutazione delle espressioni che da essa dipendono.

1.3.4 Operazioni di sollevamento

Quando la programmazione reattiva viene incorporata in un linguaggio di programmazione, come libreria o come estensione del linguaggio stesso, occorre prevedere un meccanismo per convertire gli operatori e i metodi che il programmatore è abituato ad utilizzare in modo che possano operare con le astrazioni base della RP. Quando parliamo di operazione di sollevamento, lifting operation, ci riferiamo proprio alla conversione di un operatore ordinario in una sua variante che è in grado di lavorare con valori che possono cambiare nel tempo e, quindi, al processo che permette di passare al mondo della programmazione reattiva. Lifiting è un'operazione che può essere formalizzata con la seguente definizione, assumiamo che la funzione prenda un solo parametro in ingresso:

$$lift : f(T) \longrightarrow flifted(Behaviour < T >)$$

Nella definizione, T non è un tipo di comportamento mentre, Behavior è il comportamento di un valore di tipo T. La funzione f, definita per operare su valori senza comportamento viene trasformata in una funzione che è possibile applicare ad un comportamento.
Al tempo i il valore di una funzione di sollevamento f richiamata fornendole in ingresso il comportamento di un valore di tipo T può essere definito come segue:

$$flifted(Behaviour < T >) \longrightarrow f(Ti)$$

Dove Ti denota il valore del comportamento al tempo i.
In letteratura ci sono almeno tre principali tipo di operazioni di sollevamento.

- LIFTING IMPLICITO
  In un'operazione di sollevamento implicita quando un tradizionale operatore di un linguaggio di programmazione è applicato ad un comportamento, questo automaticamente viene convertito in un operatore in grado di lavorare con il comportamento stesso.

  $$f(b1) \longrightarrow flifted(b1)$$

- LIFTING ESPLICITO
  In questo modello, il linguaggio fornisce al programmatore una serie di primitive che possono essere utilizzate per sollevare l'operazione ordinaria e quindi, per permettergli di lavorare con valori che cambiano nel tempo.

  $$lift(f)(b1) \longrightarrow flifted(b1)$$

- LIFTING MANUALE
  Il linguaggio di programmazione non prevede operatori di sollevamento; il programmatore manualmente deve acquisire il valore corrente, che nel frattempo può essere cambiato, usando gli operatori ordinari del linguaggio.

$$f(b1) \longrightarrow f(currentvalue(b1))$$

1.3.5 Multidirezionalità

Un'altra proprietà dei linguaggi di programmazione reattivi è la direzione di propagazione dei cambiamenti. I cambiamenti, infatti, possono avvenire in modalità unidirezionale o multirezionale. Un linguaggio è multidirezionale quando cambiamenti a valori derivati sono propagati anche ai valori dai quali questi derivano. Per esempio, riportiamo la formula che permette di convertire la temperatura fra Fahrenheit e Celsius:
F = (C * 1.8) +32
In questo caso, ad ogni cambiamento di F , C verrà automaticamente aggiornata e viceversa.

1.3.6 Supporto per il distribuito

Questa proprietà riguarda i linguaggi di programmazione reattivi che supportano la scrittura di programma reattivi distribuiti. I linguaggi con questa caratteristica, permettono di avere dipendenze computazionali e dipendenze fra dati distribuiti fra più nodi.

1.4 Reactive manifesto

Secondo il manifesto Reactive Manifesto (`http://www.reactivemanifesto.org`) essere reattivi significa essere di risposta (responsive), recuperabile (resilient), elastico (elastic) e guidato dai messaggi (message driven). Il manifesto nasce dalla comunità software per definire che cos'è la reattività secondo l'esperienza maturata negli anni nella realizzazione di software. Il manifesto è un breve testo ideato per definire lo stile architettonico di un sistema reattivo, non ha introdotto niente di nuovo, applicazioni reattive esistevano molto tempo prima della suo pubblicazione; un esempio, il sistema di telefonia che esiste da decenni. Esso è utile per evidenziare quali sono le pratiche migliori per creare un sistema reattivo. Passiamo ora ad una breve descrizione delle varie caratteristiche che un sistema reattivo dovrebbe prevedere (figura 1.6).

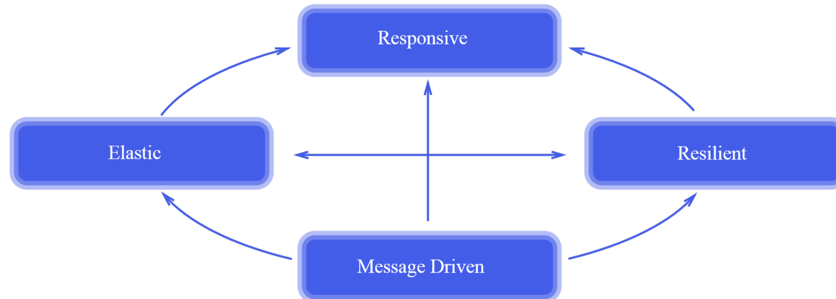

Figura 1.6: Reactive manifesto

RESPONSIVENESS (DI RISPOSTA) Quando si apre un browser e si immette un URL, ci si aspetta che la pagina che vogliamo visualizzare venga caricata in un breve periodo. Quando il caricamento richiede più di qualche millisecondo potremmo essere tentati di lasciare il sito e andare a ricercarne un altro. Se fossi stato il proprietario del sito avrei perso un cliente perché il servizio che ho fornito non è stato reattivo (di risposta). La reattività di un sistema viene misurata in base al tempo che questo impiega per rispondere ad una richiesta che ha ricevuto: un tempo di risposta breve significa che il sistema è reattivo. Un sistema può anche restituire una risposta negativa come ad esempio un messaggio di errore che informa l'utente che i dati che ha fornito in input sono sbagliati ma, indipendentemente da questo, il tempo che il sistema impiega per elaborare la richiesta e fornire una risposta in sistemi reattivi deve essere ragionevole. Definire un tempo ragionevole di risposta è difficile poiché questo dipende dal contesto e dal sistema che stiamo misurando. Per una applicazione client con un bottone si assume che il tempo che l'applicazione dovrebbe impiegare per gestire la sua pressione sia di qualche millisecondo ma, ad esempio, per un servizio web che ha bisogno di fare calcoli pesanti uno o due secondi possono essere considerati ragionevoli. Quando si progetta un'applicazione si dovrebbe, per ogni operazione che si ha, definire un tempo massimo per completarla e per fornire un risposta all'utente. Essere di risposta e, quindi rispondere nel minor tempo possibile, è l'obiettivo principale dei sistemi reattivi.

RESILIENCY (RECUPERABILE) Il nostro sistema potrebbe fallire, ad esempio, si disconnette dalla rete, si rompono dei dischi rigidi, si hanno problemi con l'elettricità ecc... Un sistema recuperabile è un sistema in grado di rimanere reattivo anche in caso di rotture cioè, è in grado di gestirle senza

che l'utente possa accorgersene poiché continua ad ottenere le risposte che desidera. Il modo attraverso il quale si garantisce recuperabilità cambia da sistema a sistema, un esempio, aggiungere più server in modo che se uno si dovesse bloccare, le richieste possono essere gestite dagli altri. In generale, un modo per aumentare la recuperabilità di un sistema è quello di evitare singoli punti di rottura e quindi, progettare e creare ogni parte dell'applicazione in modo più isolato possibile dalle altre.

ELASTIC (ELASTICO) La nostra applicazione sarà utilizzata da un grande numero di utenti ciascuno dei quali farà richieste al sistema che possono provocare un carico elevato. Ogni componente del sistema ha un limite al livello di carico che può affrontare, e quando questo viene superato, le richieste non potranno più essere soddisfatte e il sistema stesso potrebbe bloccarsi. Questa situazione di crescente carico può anche essere un Distributed Denial of Service (DDoS) che il sistema sta vivendo. Per evitare ciò il sistema deve essere elastico cioè espandere le sue istanze quando il carico aumenta e rimuoverle quando il carico diminuisce.

MESSAGE DRIVEN (GUIDATO DAI MESSAGI) Se osserviamo le caratteristiche di risposta, recuperabilità e elasticità di cui abbiamo parlato poco prima, possiamo dire che il nostro obiettivo è quello di avere un sistema "di risposta", la recuperabilià è il modo con cui garantiamo la reattività e, l'elasticità è un metodo per rendere il sistema recuperabile. Il pezzo mancante del puzzle dei sistemi reattivi è il modo con cui le parti comunicano tra loro per garantire la reattività del sistema. Lo scambio asincrono di messaggi è ciò che meglio si adatta alle nostre esigenze, in quanto ci permette di controllare il livello di carico su ogni componente; permette di inoltrare i messaggi alla giusta destinazione e di rispedire i messaggi in caso di crash di uno dei componenti. L'utente non avrà bisogno di conoscere l'architettura interna del sistema ma solo il tipo di messaggi che questo è in grado di gestire. Un sistema guidato dai messaggi (message driven) rende possibile tutti gli altri concetti visti precedentemente. La figura (ficura 1.7) mostra come tale approccio, utilizzando una coda di messaggi, aiuta a ripartire il carico di lavoro e, come permette di garantire elasticità e recuperabilità.

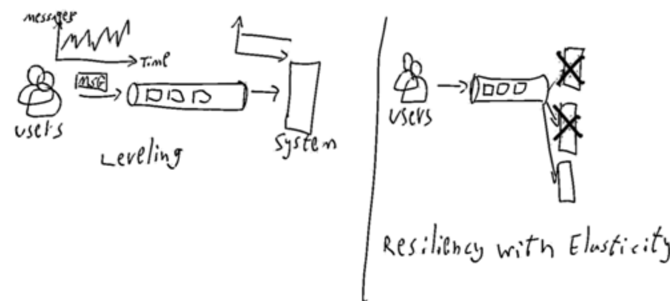

Figura 1.7: sulla sinistra possiamo vedere che sebbene i messaggi arrivino ad alta frequenza, l'elaborazione da parte del sistema avviene alla stessa frequenza. Sulla destra, invece, possiamo notare come il sistema riesce a far fronte ad un crescente numero di richieste e ad eventuali guasti senza che l'utente se ne possa accorgere poiché continuerà a ricevere risposte.

In questa tesi si affronteranno le estensioni reattive di diversi linguaggi di programmazione e piattaforme, librerie che non si occupano dello scambio di messaggi fra applicazioni e server ma piuttosto, prevedono meccanismi per la gestione dei messaggi quando arrivano e del passaggio di questi attraverso la catena di operazioni interne all'applicazione. Esse sono quindi utili per gestire eventi e messaggi che le applicazione devono elaborare. La relazione fra il Reactive Manifesto e tali librerie può essere rappresentata nella figura che segue (figura 1.8)

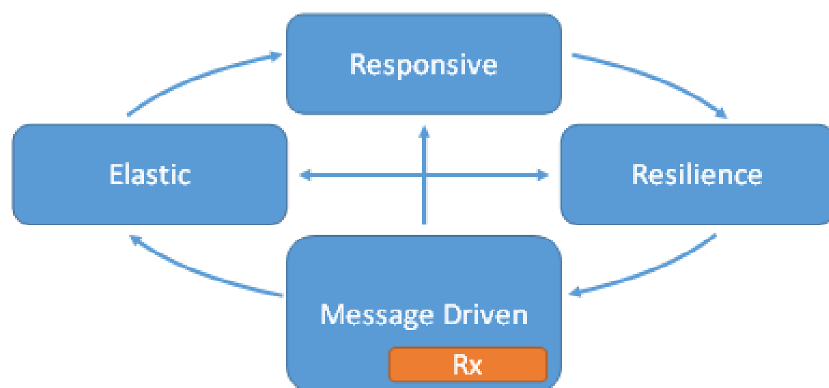

Figura 1.8: Relazione del Reactive manifesto con la libreria Rx(Reactive Extension).

1.5 Classificazione dei linguaggi reattivi

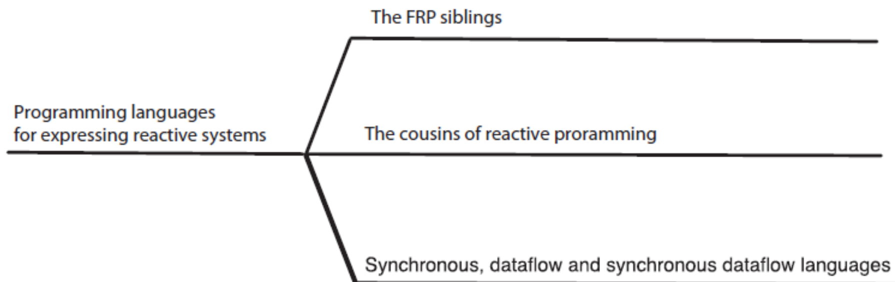

Figura 1.9: Classificazione dei linguaggi nella programmazione reattiva

E' possibile suddividere i linguaggi di programmazione reattiva in tre diverse categorie: i fratelli della programmazione reattiva funzionale FRP (FRP siblings), i cugini della programmazione reattiva (cousins of reactive programming) e, linguaggi che si occupano di gestire flussi sincroni di dati (figura 1.9).

The FRP siblings I linguaggi di programmazione reattivi di questa categoria prevedono le astrazioni base, bahaviors e events, per rappresentare valori continui e discreti e forniscono primitive per la loro gestione e manipolazione. Nella FRP, quando i valori degli argomenti di una funzione variano, automaticamente tali cambiamenti vengono propagati e la funzione rivalutata. La FRP permette al programmatore di esprimere programmi reattivi in modo dichiarativo. Per esempio, un semplice programma funzionale che disegna un cerchio nello schermo nella posizione corrente del mouse può essere espresso come `(draw-circle mouse-x mouse-y)`. In questa espressione, quando il valore di `mouse-x` o `mouse-y` cambiano, la funzione `draw-circle` viene automaticamente ricalcolata e aggiornata la posizione del cerchio all'interno dello schermo. Elliott e Hundak [1997] identificarono come vantaggio chiave del paradigma FRP: *clarity, ease of construction, composability, and clean semantics.* La FRP è stata introdotta in Fran[Elliott e Hudak 1997] come linguaggio appositamente progettato per lo sviluppo di grafica interattiva e animazioni in Haskell.

Da allora, le idee della FRP sono state esplorate in differenti linguaggi tra cui Yampa, FrTime, Flapjax, ecc.

L'innovativa ricerca sulla programmazione reattiva è stato effettuata principalmente nel contesto della FRP. Non è quindi sorprendente che un gran numero di linguaggi evolvere attorno alla nozione della FRP.

Nella tabella (tabella 1.1) è possibile vedere parte dei linguaggi reattivi appartenenti a questa categoria.

Tabella 1.1: Linguaggi reattivi fratelli della programmazione reattiva funzionale (FRP)

Language	Host Language
Fram [Elliott and Hudak 1997]	Haskell
Yampa [Hudak et al. 2003]	Haskell
Frappé [Courtney 2001]	Java
FrTime [Cooper and Krishnamurthi 2006]	Racket
NewFran [Elliott 2009]	Haskell
Flapjax [Meyerovich et al. 2009]	Javascript
Scala.React [Maier et al. 2010]	Scala
AmbientTalk/R [Carreton et al. 2010]	AmbientTalk

The cousin of reactive programming I linguaggi di programmazione reattivi di questa categoria non prevedono le astrazioni base quali eventi e comportamenti ma, prevedono il supporto per la propagazione automatica dei cambiamenti di stato e altre caratteristiche della programmazione reattiva come la proprietà di evitare glitch. In tali linguaggi le astrazioni per rappresentare la variazione dei valori nel tempo non è integrata con il linguaggio host, le operazioni di lifting o trasformazione devono essere fatte manualmente dal programmatore. Tali linguaggi vengono detti cugini della programmazione reattiva, in tabella (tabella 1.2) è possibile trovarne alcuni esempi. Siccome tali linguaggi non prevedono operatori per combinare eventi e comportamenti come merge, switch ecc.. è molto più difficile scrivere applicazioni o funzioni rispetto a linguaggi considerati fratelli della FRP.

Tabella 1.2: Linguaggi reattivi 'cugini' della programmazione reattiva

Language	Host Language
Cell [Tilton 2008]	CLOS
Lamport Cells [Miller 2003]	E
SuperGlue [McDirmid e Hsieh 2006]	Java
Trellis [Eby 2008]	Python
.NET Rx [Hamilton e Dyer 2010]	C sharp.NET

The FRP siblings vs The cousin of reactive programming La differenza principale fra i linguaggi di programmazione reattivi classificati come fratelli della FRP e i cugini della programmazione reattiva è che i primi forniscono

astrazioni per rappresentare la variazione dei valori nel tempo e interagiscono con il linguaggio host mediante operazioni di lifting implicite o esplicite. I secondi, invece, forniscono 'contenitori' o 'celle' dove i valori possono cambiare nel tempo, il programmatore deve manualmente memorizzare i valori in queste celle e manualmente estrarli. Il sistema si prende carico però della corretta propagazione dei cambiamenti attraverso la rete di 'celle'.

In pratica questo significa che nei linguaggi fratelli della FRP i valori che cambiano nel tempo possono essere utilizzati nelle espressioni ordinarie (anche se con operazioni di sollevato esplicite in alcuni casi) e la gestione delle dipendenze avviene automaticamente, nei linguaggi "cugini", invece, è richiedono del codice dedicato per codificare queste dipendenze.

Capitolo 2

Esempi di tecnologie: RxJS e RxPHP

In questo capitolo varrà fornita una panoramica generale della libreria Reactive Extension (Rx) successivamente, verranno approfondite le estensione reattive previste per Javascript (RxJS) e per PHP (RxPHP). La scelta è ricaduta su queste due estensioni poiché ho deciso di testare il paradigma di programmazione reattivo in una web app sviluppata appunto, utilizzando PHP per il back-end e Javascript per parte del front-end.

2.1 Reactive Extension

"A library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators"

Reactive Extension(Rx) è una libreria sviluppata da Microsoft nel 2009 per lavorare facilmente con stream asincroni di eventi e dati. E' stata progettata da Erik Meijer e Brian Beckman e si ispira allo stile della programmazione funzionale. In Rx ogni stream viene rappresentato con un oggetto Observable e può essere creato da qualsiasi sorgenti come da eventi dell'interfaccia grafica, da tweet, da delle webSocket, da degli indici di borsa ecc ... Tali stream possono essere interrogati con operatori LINQ ed è possibile controllare la concorrenza attraverso gli Schedulers; per questo Rx è spesso definita come:

Rx = Observables + LINQ + Schedeulers

Tali concetti varranno approfonditi in seguito.

Tabella 2.1: Collezioni di dati nella programmazione asincrona

	singolo elemento	collezione
sincrono	$TgetObject()$	$Iterable < T > getCollection()$
asincrone	$Future < T > getObject()$	$Observable < T > getCollection()$

Observable Rx ci permette di lavorare con gli eventi derivanti dall'interfaccia grafica o con chiamate asincrone alla stessa maniera con cui operiamo sulle collezione. Trattiamo questi eventi e chiamate asincrone come un flusso che viene modificato e gestito man mano che si genera ad opera della classe Observable. Cerchiamo di inquadrarlo meglio. Quando ci serve avere un oggetto da un metodo, definiamo il nostro metodo in questo modo:

$$TgetObject()$$

mentre se ci serve una collezione scriviamo:

$$Iterable < T > getCollection()$$

Ora se ci spostiamo nel campo dell'esecuzione asincrona e vogliamo gestire un oggetto che sarà disponibile successivamente, andiamo a realizzare un metodo con questo stile:

$$Future < T > getObject()$$

Ci manca un'ultima combinazione: una collezione i cui elementi saranno disponibili in futuro ma non necessariamente tutti nello stesso istante. La soluzione per questo caso è la classe Observable proposta dalla libreria Reactive X

$$Observable < T > getCollection()$$

Un Observable è un oggetto che emette zero o più elementi per poi terminare con successo oppure durante il flusso degli elementi si interrompe a causa di un errore. In altre parole, un Observable può essere definito come una sorgente di dati osservabile che può inviarli a chiunque sia interessato. Gli Observer sono, invece, gli osservatori cioè coloro che rimangono in ascolto della sorgente dati e che reagiscono ad ogni nuovo valore da questa emesso.

L'Observable quando ha a disposizione un nuovo elemento notifica tutti gli observer registrati invocando un loro metodo. Attraverso il metodo Subscribe è possibile connettere un Observer ad un Observable.

```
interface IObservable<T>{
   IDisposable Subscribe(IObserver<T>);
}
```

Listato 2.1: Interfaccia di un osservabile

Observer L'observer rimane in ascolto di un observable. Quando un evento avviene in una observable, esso richiama il metodo corrispondente in ogni suo osservatore. L'Observer potrà avere i seguenti metodo (vedere 2.2):

- onNext
  Tutte le volte che un Observable emette un nuovo elemento chiamerà questo metodo, il quale prende in ingresso proprio l'oggetto emesso dell'Obsevable.

- OnComplete
  Questo metodo viene richiamato dall'Observable dopo l'ultimo OnNext per indicare che non emetterà altri elementi e cioè, che è terminato correttamente e senza alcun errore.

- onError
  Questo metodo interrompe l'Observable cioè non potranno seguire altre chiamate di onNext o onComplete e, prende in ingresso un parametro che riporta la causa dell'errore.

Un Observable potrà chiamare zero, una o più volte il metodo onNext dell'Obsever ma una, ed una sola volta, il metodo onComplete o onError poiché, dopo una di queste due chiamate, l'Observer termina.

```
interface IObserver<T>{
   void onNext(T value);
   void onError(Exception error);
   void onCompleted();
}
```

Listato 2.2: Interfaccia di un osservatore

Rx pattern Un observable ha origine dalla combinazione di due pattern: Observer e Iterator (the Gang of Four's pattern). La sequenza osservabile o, semplicemente, l'osservabile emette i suoi valori in ordine, come un iterator, ma non sono i suo consumatori a richiedere il prossimo valore: l'osservabile 'spinge' i valori ai consumatori non appena disponibili. L'osservabile dunque,

ha un ruolo simile al produttore del pattern observer il quale emette valori e gli inoltra ai sui ascoltatori mentre, gli osservatori sono l'equivalente degli ascoltatori del pattern. l'Observable estende il pattern observer ma ci sono delle differenze importanti, infatti l'Observable:

- segnala all'Observer che sta terminando chiamando il metodo onComplete.
- Informa l'Observer che è avvenuto un errore chiamando il metodo onError.

Disposable Quando un observer sottoscrive un osservable ottiene un oggetto disposable. Attraverso tale istanza l'osservatore può eliminare la sua sottoscrizione alla sequenza(cancellazione esplicita).

```
interface IDisposable{
   void Dispose();
}
```

Listato 2.3: Interfaccia di un disposable

Hot e Cold observable Ciò che differenzia questi observable è il momento in cui iniziano ad emettere la sequenza di valori. Un Hot Observable inizia ad emettere elementi non appena viene creato perciò, quando un observer lo sottoscrive, riceverà notifica dei soli elementi emessi dopo la sottoscrizione. Un Cold Observables, invece, aspetta la sottoscrizione da parte di un observer prima di emettere elementi; ogni observer riceverà l'intera sequenza dell'observable (cioè dal primo elemento) indipendentemente dal momento di sottoscrizione.

Operatori Ogni estensione reattiva dei linguaggi di programmazione fornita da Rx prevede una serie di operatori ma molti di questi sono previsti in tutte, simile è la funzione che svolgono sull'observable di partenza così come il nome dell'operazione stessa. La maggior parte degli operatori opera su un osservabile e restituisce un nuovo osservabile. Ciò consente di applicare questi operatori uno dopo l'altro, in catena. Ogni operatore della catena modifica l'observable che deriva dall'esecuzione del precedente operatore. E' possibile, dunque, interrogare tali sequenza asincrone di dati usando generici operatori LINQ.

Gli operatori previsti da Rx possono essere raggruppati in categorie, vediamo alcune:

- operatori per creare nuovi observable
- operatori che permettono di trasformare gli elementi emessi dall'observable
- operatori per selezionare parte degli elementi emessi dalla sequenza
- operatori per combinare più observable
- .....

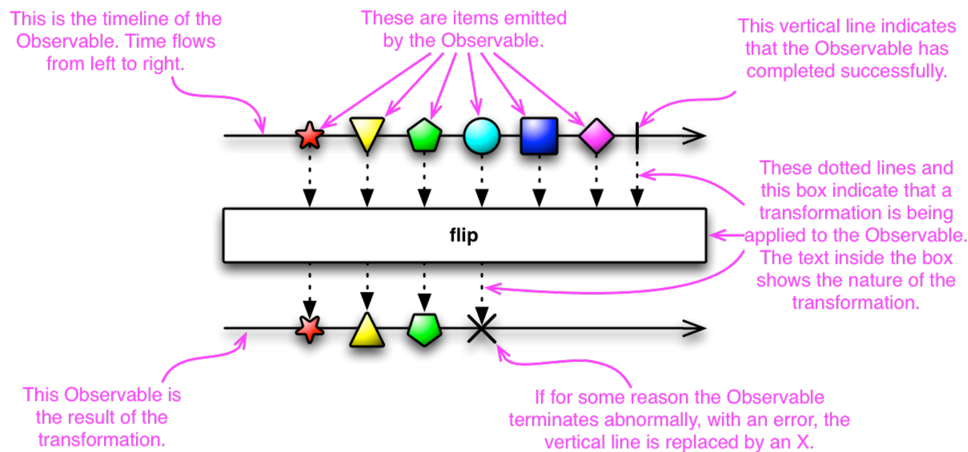

Figura 2.1: Diagramma a biglie o marble diagram

Marble diagram "Un'immagine vale più di mille parole". Questo è vero quando parliamo di programmazione reattiva e, in particolare di Rx. I diagrammi a biglie (Marble diagram) usano assi orizzontali per raffigurare gli observable, ogni evento o dato emesso dalla sequenza viene rappresentato con un simbolo. Generalmente il valore del dato o dell'evento viene scritto all'interno del simbolo o come nota accanto. In particolare, gli assi orizzontali simboleggiano la linea del tempo che scorre da sinistra verso destra, la distanza fra un simbolo e l'altro mostra un'approssimazione del tempo intercorso fra i due eventi. Per rappresentare il fatto che l'Observable è stato completato viene utilizzato il simbolo — , mentre in caso di errore, una X. Il rettangolo mostra l'operazione eseguita sull'observable mentre le frecce verticali tratteggiate la relazione fra l'elemento di partenza e l'elemento corrispondente nell'observable ottenuto come risultato dell'operatore. La documentazione di Rx fa uso estensivo di tali diagrammi.

Scheduler Se si vuole introdurre il multithreading nella cascata di operatori applicati all'Observable, è possibile farlo istruendo gli operatori (o Obervable ad hoc) per operare su particolari Scheduler. Alcuni operatori di ReactiveX hanno varianti che prendono uno scheduler come un parametro. Questo comunica all'operatore di fare parte o totalità del suo lavoro su un particolare scheduler. Per default, un osservabile e la catena di operatori che si applica ad esso farà il suo lavoro, e notificherà i suoi osservatori, sullo stesso thread su cui è chiamato il metodo Sottoscrivi. L'operatore SubscribeOn, ad esempio, variante del metodo subscribe, permette di specificare uno scheduler che l'Observable utilizzerà per inviare notifiche ai propri osservatori.

Subject Un subject è una sorta di ponte o proxy disponibile in alcune implementazioni di Rx che agisce sia come osservatore che come osservabile. Poiché si tratta di un osservatore, può sottoscrivere (subscribe) uno o più observable e, poiché si tratta di un osservabile, può emettere nuovi elementi o gli elementi che gli vengono inviati dagli Observable che ha precedentemente sottoscritto in veste di observer.

Oggi Rx supporta decine di linguaggi per ciascuno dei quali viene fornita una sua estensione reattiva ma tutti con le stesse astrazioni base di cui supra si è discusso quali Observable, Observer, Scheduler ecc......

2.2 RxJS

Nel paragrafo precedente è stata fornita un panoramica generale della libreria Rx, di seguito verrà trattata nel dettaglio l'estensione reattiva fornita per il linguaggio Javascript (RxJS).

Osservabili e osservatori Per ricevere notifiche da parte dell'Observable quando ha nuovi valori a disposizione occorre invocare il metodo subscribe passandogli come argomento proprio l'observer interessato a tali valori. il metodo subscribe restituisce un oggetto disposable che permette di gestire la sottoscrizione infatti, attraverso questo, è possibile scollegare l'observer dalla sequenza osservabile quando non vuole più ricevere valori. In RxJS non è necessario svincolare esplicitamente l'observer dalla sorgente poiché questo avviene automaticamente dopo ogni chiamata del metodo onError o onComplete. Vediamo un esempio.

```
var source = Rx.Observable.range(1, 5);
```

```
var subscription = source.subscribe(
    function (x) { console.log('onNext: ' + x); },
    function (e) { console.log('onError: ' + e.message); },
    function () { console.log('onCompleted'); });
```

Listato 2.4: Sequenza osservabile di cinque interi in RxJS

Nel frammento di codice (2.4) viene creata una sequenza osservabile di 5 interi, partendo da 1, e sottoscritta da un osservatore. Al metodo subscribe vengono passate tre funzioni che rappresentano rispettivamente, la funzione onNext, onError e onComplete dell'observer. In questo caso l'osservatore si limita a stampare su console gli elementi ricevuti dall'osservabile.
L'output sarà il seguente: onNext:1, onNext:2, onNext:3, onNext:4, onNext:5, onCompleted

Vediamo ora come creare un observable da una mappa (codice 2.5).

```
var map = new Map([['key1', 1], ['key2', 2]]);
var source = Rx.Observable.from(map);

var subscription = source.subscribe(
  function (x) { console.log('onNext: %s', x); },
  function (e) { console.log('onError: %s', e); },
  function () { console.log('onCompleted'); });

Output:
onNext: key1, 1
onNext: key2, 2
onCompleted
```

Listato 2.5: Sequenza osservabile generata a partire da una mappa in RxJS

Sequenze osservabili di eventi Si è mostrato come è possibile creare observable da collezioni ora, vediamo come sia possibile farlo anche con gli eventi, ad esempio, generati dall'utente durante l'interazione con l'interfaccia grafica. L'esempio che segue (codice 2.6) mostra come creare una sequenza di eventi 'movimento del mouse' attraverso l'operatore fromEvent: l'evento viene trasformato in un oggetto di prima classe. Tutte le volte che il mouse cambia la sua posizione l'osservatore viene notificato e richiamato il metodo onNext passando come parametro l'evento stesso. Successivamente dall'observable `allMoves` vengono creati altre due sequenza che si occupano rispettivamente di filtrare gli eventi che avvengono sulla parte destra e sinistra dello schermo. Nessuno dei due però, modifica la sequenza iniziale `allMoves` poiché ogni operatore applicato su questa produce una nuova sequenza.

```
var allMoves = Rx.Observable.fromEvent(document, 'mousemove');

var subscription = allMoves.subscribe(function (e) {
   console.log(e.clientX + ', ' + e.clientY);
});

var movesOnTheRight = allMoves.filter(function(e){
   return e.clientX > window.innerWidth / 2;
});

var movesOnTheLeft = allMoves.filter(function(e){
   return e.clientX < window.innerWidth / 2;
});

movesOnTheRight.subscribe(function(e){
   console.log('Mouse is on the right:', e.clientX);
});

movesOnTheLeft.subscribe(function(e){
   console.log('Mouse is on the left:', e.clientX);
});
```

Listato 2.6: Sequenze osservabili di eventi

Sequenze osservabili di callback e promesse E' abbastanza semplice convertire promesse in una sequenza osservabile. Per realizzare tale trasformazione utilizziamo il metodo Rx.Observable.fromPrimise che si occupa di gestire la promessa sia in caso di successo che in caso di fallimento. Nell'esempio seguente per usare la promessa ci avvaliamo del libreria RSVP.

```
// Create a promise which resolves 42
var promise1 = new RSVP.Promise(function (resolve, reject) {
    resolve(42);
});

var source1 = Rx.Observable.fromPromise(promise1);

var subscription1 = source1.subscribe(
  function (x) { console.log('onNext: %s', x); },
  function (e) { console.log('onError: %s', e); },
  function () { console.log('onCompleted'); });
```

```
=> onNext: 42
=> onCompleted
```

Listato 2.7: Sequenza osservabile generata da una promessa

Operatori (in catena) sulle sequenze osservabili Come già specificato, la maggior parte degli operatori prende come input una sequenza osservabile, esegue su questa una determinata funzione e riporta come output un'altra sequenza: proprio per questo, possiamo creare catene di operatori. Di seguito ne verranno mostrati alcuni.

Concat L'operatore concat (figura 2.2) prende in ingresso un numero variabile di observable (o un array di observable) e gli concatena nello stesso ordine con cui vengono forniti.

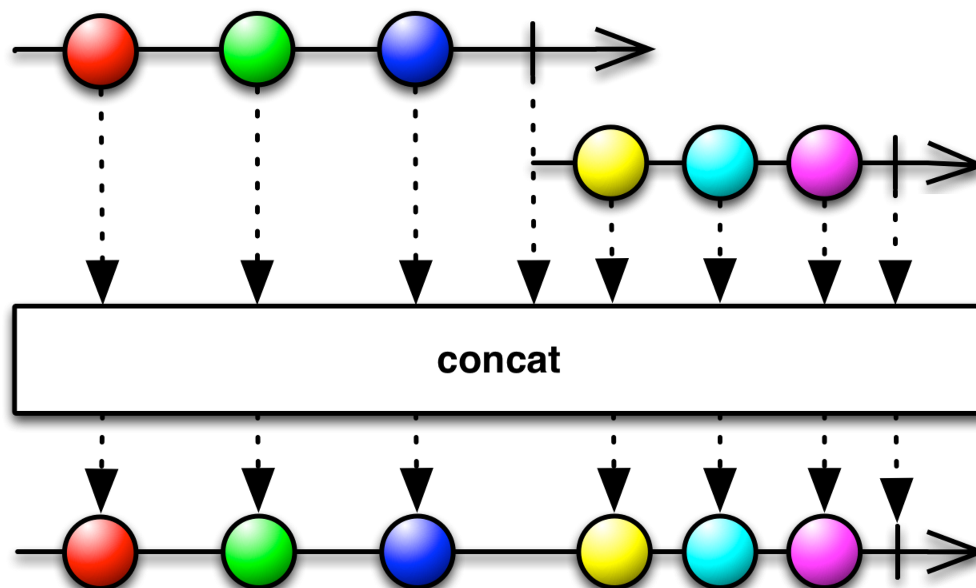

Figura 2.2: Operatore Concat

```
var source1 = Rx.Observable.range(1, 3);
var source2 = Rx.Observable.range(1, 3);

source1.concat(source2)
   .subscribe(function (x) { console.log(x); });
```

Listato 2.8: Esempio di utilizzo dell'operatore concat in RxJS

l'output sarà: 1,2,3,1,2,3 poiché la seconda sequenza (`source2`)verrà attivata solo dopo la terminazione della prima (`source1`).

Merge L'operatore merge (figura 2.3) prende in ingresso una sequenza di observable e unisce i loro elementi non appena vengono emessi da questi.

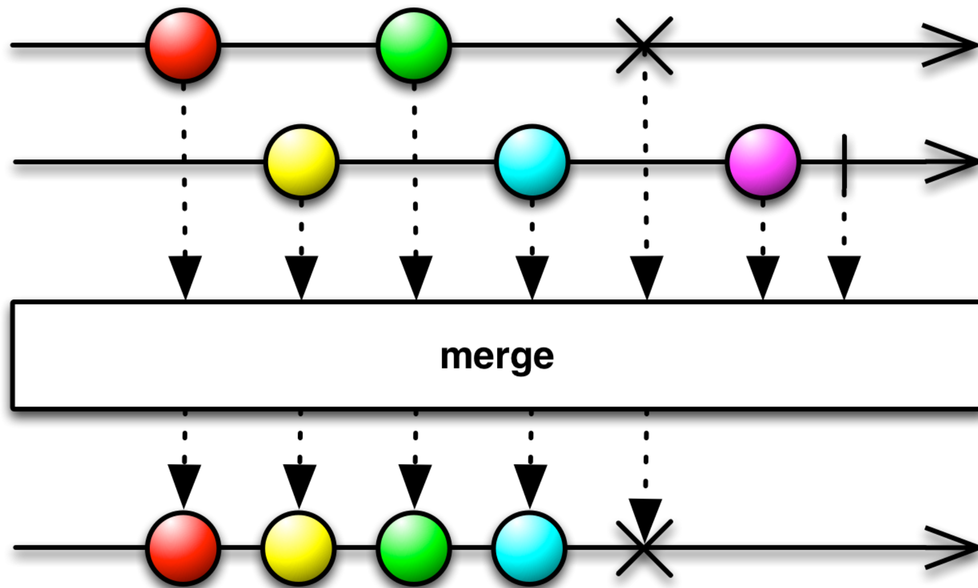

Figura 2.3: Operatore Merge

```
var source1 = Rx.Observable.range(1, 3);
var source2 = Rx.Observable.range(1, 3);

source1.merge(source2)
   .subscribe(function (x) { console.log(x); });
```

Listato 2.9: Esempio di utilizzo dell'operatore merge in RxJS

l'output sarà: 1,1,2,2,3,3 poiché le sequenze vengono attivate nello stesso istante e i valori aggiungi alla sequenza di output non appena emessi da quelle di partenza.

Map L'operatore map (figura 2.4) è in grado di trasformare ogni elemento della sequenza in un'altra forma, come parametro di ingresso accetta la funzione di trasformazione.

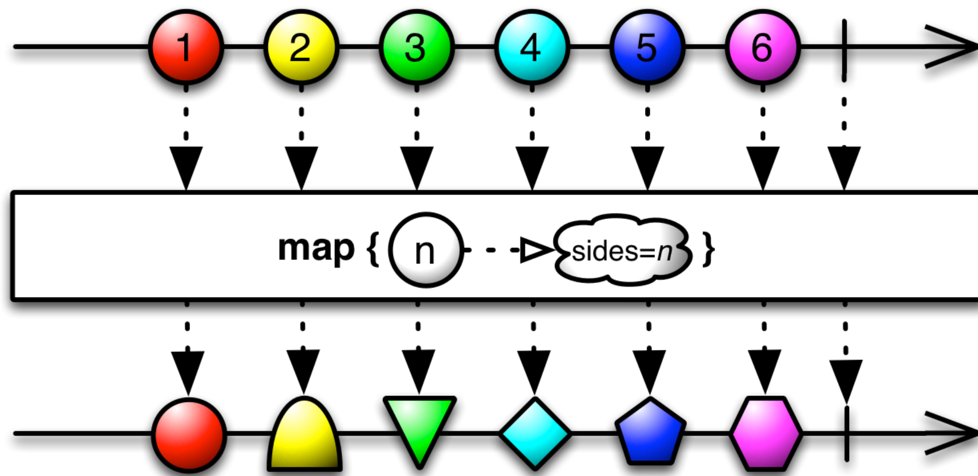

Figura 2.4: Operatore Map

Nell'esempio che segue (listato 2.10) attraverso l'operatore map provvediamo a trasformare una sequenza di stringhe in una sequenza di interi che rappresentano la loro lunghezza.

```
var array = ['Reactive', 'Extensions', 'RxJS'];

var seqString = Rx.Observable.from(array);

var seqNum = seqString.map(function (x) { return x.length; });

seqNum
   .subscribe(function (x) { console.log(x); });
```

Listato 2.10: Esempio di utilizzo dell'operatore map in RxJS

L'output sarà: 8,10,4
In questo caso la funzione di trasformazione prende in ingresso solo un valore (x) cioè il valore correntemente emesso dalla sequenza ma può anche avere altri due parametri rispettivamente, l'indice dell'elemento all'interno della sequenza e l'Observable che lo ha emesso.

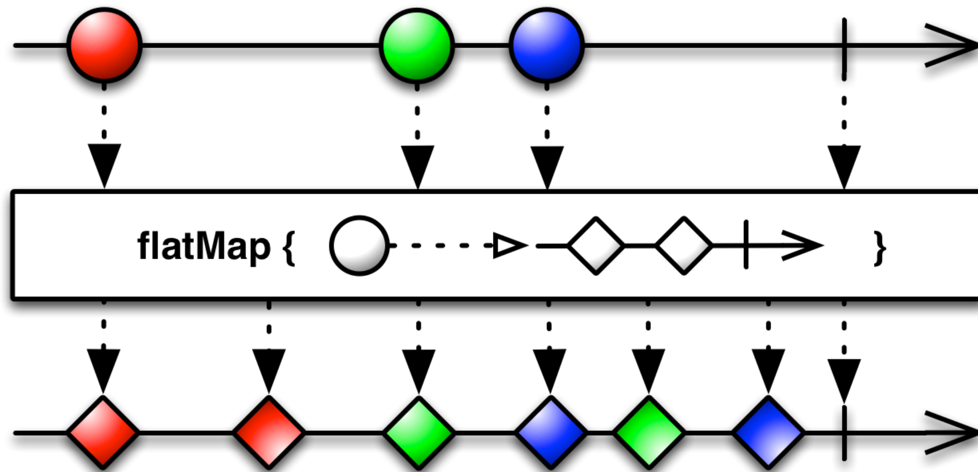

Figura 2.5: Operatore FlatMap

FlatMap L'operatore FlatMap (figura 2.5) trasforma ogni elemento emesso dall'Observable in un nuovo Observable, uniti successivamente per formarne uno unico. Ci sono diverse versioni di questo operatore quella che viene mostrata di seguito prende come input una funzione di trasformazione che restituisce anch'essa un observable.

```
var source = Rx.Observable
    .range(1, 2)
    .selectMany(function (x) {
        return Rx.Observable.range(x, 2);
    });

var subscription = source.subscribe(
    function (x) { console.log('Next: ' + x); },
    function (err) { console.log('Error: ' + err); },
    function () { console.log('Completed'); });
```

Listato 2.11: Esempio di utilizzo dell'operatore flatMap in RxJS

L'output sarà il seguente:Next: 1, Next: 2, Next: 3, Next: 4, Completed.

Filter L'operatore Filter (figura 2.6) emette solo gli elementi di un observable che soddisfano il predicato di test. La funzione di predicato accetta in ingresso tre parametre: l'elemento emesso dalla sequenza, l'indice dell'elemento (partendo da 0) e l'observable stesso.

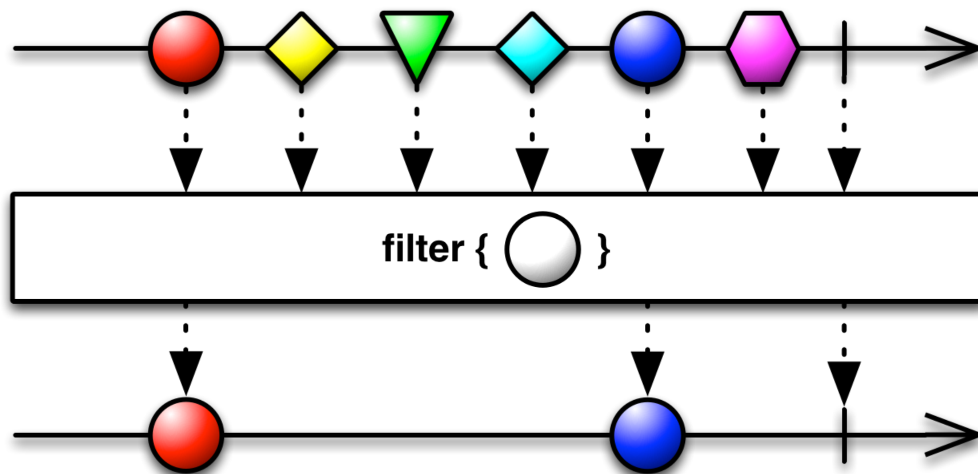

Figura 2.6: Operatore Filter

```
var source = Rx.Observable.range(0, 5)
  .filter(function (x, idx, obs) {
    return x % 2 === 0;
  });

var subscription = source.subscribe(
  function (x) { console.log('Next: %s', x); },
  function (err) { console.log('Error: %s', err); },
  function () { console.log('Completed'); });
```

Listato 2.12: Esempio di utilizzo dell'operatore filter in RxJS

L'output sarà: Next:1, Next:2, Next:3, Completed

Pipeline di operatori Viene mostrato (listato 2.13) un esempio di pipeline di operatori in RxJs.

```
stringObservable
   .map(function(str){
      return str.toUpperCase();
   })
   .filter(function(str){
      return /^[A-Z]+$/.test(str);
   })
   .take(5)
```

```
  .subscribe(function(str){
    console.log(str);
  });
```

Listato 2.13: Esempio pipeline di operatori in RxJS

La gestione dell'errore Uno dei compito più difficili nella programmazione asincrona è la gestione degli errori. Diversamente dalla programmazione imperativa non possiamo usare semplicemente l'approccio try/catch/finally. Il contratto observer prevede il metodo onError per gestire eventuali errori generati dalla sequenza osservabile ma, potremmo avere la necessità di far finta che non sia mai accaduto e,ad esempio, effettuare le seguenti operazioni:

- far sparire l'errore e passare ad un observable di backup per continuare la sequenza,
- far sparire l'errore ed emettere un elemento predefinito,
- far sparire l'errore e subito, cercare di riavviare l'observable che ha fallito,
- ....

Riportiamo alcuni esempi che prevedo meccanismi per gestire errori.

OnErrorResumeNext Questo operatore specifica che, quando a run-time, avviene un errore il controllo deve passare al valore successivo nella sequenza osservabile e proseguire la sua esecuzione da questo punto (esempio 2.14).

```
var source = Rx.Observable.onErrorResumeNext(
  Rx.Observable.just(42),
  Rx.Observable.throw(new Error()),
  Rx.Observable.just(56),
  Rx.Observable.throw(new Error()),
  Rx.Observable.just(78)
);

var subscription = source.subscribe(
  function (data) {
    console.log(data);
  }
);
```

Listato 2.14: OnErrorResumeNext

La sequenza emetterà i seguenti valori: 42, 56, 78

Retry Con l'operatore retry è possibile provare un'operazione un determinato numero di volte prima che venga generato un errore. Questo può essere utile quando è necessario ottenere dati da una risorsa che può avere guasti intermittenti a causa del carico o per qualsiasi altra questione. Nell'esempio che segue (codice 2.15) si cerca di ottenere dati da un url e solo dopo 3 fallimenti verrà notificato l'observer dell'errore.

```
var source = get('url').retry(3);

var subscription = source.subscribe(
  function (data) {
    console.log(data);
  },
  function (err) {
    console.log(err);
  }
);
```

Listato 2.15: Retry

Subjects Attraverso un esempio viene mostrato l'utilizzo dei subjects in RxJS. Creiamo un subject e lo usiamo una volta, come observer per sottoscrivere una sequenza che produce un intero ogni secondo l'altra, come observable e quindi come sorgente dati.

```
// Observable che produce un intero ogni secondo
var source = Rx.Observable.interval(1000);

var subject = new Rx.Subject();

var subSource = source.subscribe(subject);

var subSubject1 = subject.subscribe(
    function (x) { console.log('Value published to observer 1: ' +
       x); },
    function (e) { console.log('onError: ' + e.message); },
    function () { console.log('onCompleted'); });
```

```
setTimeout(function () {
    // Clean up
    subject.onCompleted();
    subSubject1.dispose();
}, 5000);

Value published to observer 1: 0
Value published to observer 1: 1
Value published to observer 1: 2
Value published to observer 1: 3
onCompleted
```

Listato 2.16: Subjects

Tale esempio (codice 2.16) mostra la natura di un subject di fungere da proxy.

RxJS prevede diverse varianti di Subject come:

- **ReplaySubject**
  Tale subject (figura 2.7) memorizza tutti i valori che ha pubblicato pertanto, quando un observer lo sottoscrive, indipendentemente dal momento, riceve automaticamente tutta la sequenza. Per evitare un consumo eccessivo di memoria è possibile limitare i dati emessi dalla sequenza memorizzati fornendo al costruttore del subject una dimensione per il buffer oppure un finestra temporale.

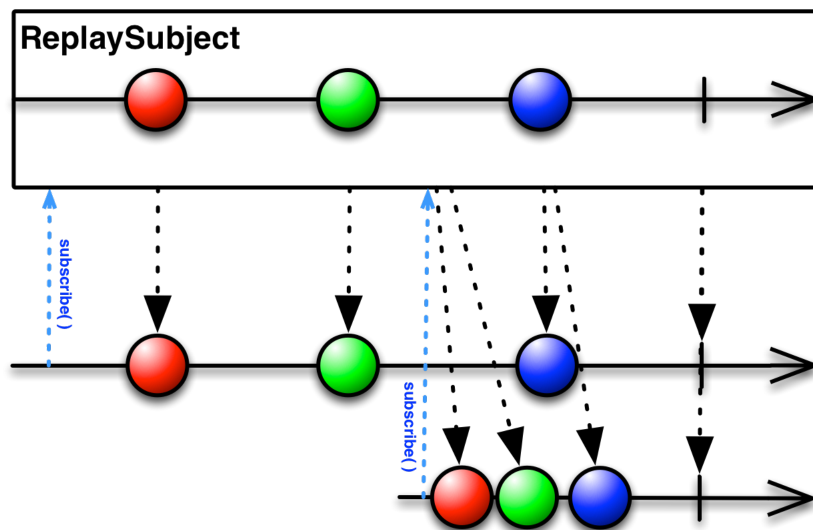

Figura 2.7: ReplaySubject

```
var subject = new Rx.ReplaySubject();

subject.onNext();

subject.subscribe(function(n){
   console.log('Received value:', n)
});

subject.onNext(2);
subject.onNext(3);

output:
Received value:1
Received value:2
Received value:3
```

Listato 2.17: ReplaySubject in RxJs

Lo stesso esempio (codice 2.17) con un classico subject avrebbe prodotto il seguente output: Received value:2, Received value:3.

- **BehaviorSubject**

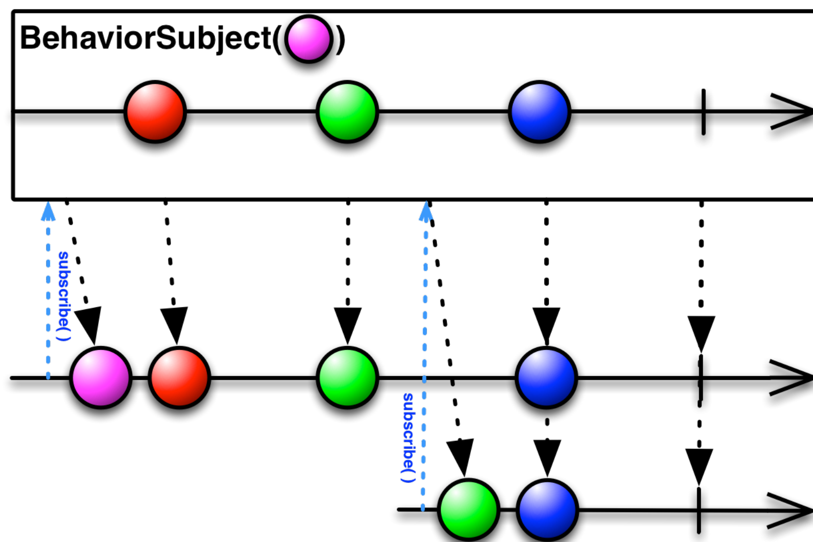

Figura 2.8: BehaviorSubject

Molto simile a ReplaySubject ad eccezione del fatto che memorizza solo l'ultimo valore che ha emesso in veste di observable. Esso (figura 2.8) richiede un valore predefinito al momento dell'inizializzazione, valore che viene inviato agli osservatori quando ancora il subject non ha emesso alcun elemento. Ciò significa che ogni observer riceverà automaticamente un valore subito dopo la sottoscrizione (ultimo valore emesso) a meno che il Subject non sia già terminato.

```
var subject = new Rx.BehaviorSubject('Waiting for content');

subject.subscribe(
   function(result){
      document.body.textContent = result.response || result;
   }
   function(err){
      document.body.textContent = 'There was an error';
   }
);

Rx.DOM.get('/remote/content').subscribe(subject);
```

Listato 2.18: BehaviorSubject in RxJs

Nel frammento di codice (codice 2.18) viene inizializzato un behavior-Subject con un placeholder o valore iniziale visualizzato nel corpo del documento finché non si ottiene la risposta desiderata da remoto.

- **AsyncSubject**

```
var delayRange = Rx.Observable.range(0,5).delay(500);

var subject = new Rx.AsyncSubject();

delayRange.subscribe(subject);

subject.subscribe(
   function onNext(item){console.log('value:', item);},
   function onError(err){console.log('error:', err);},
   function onCompleted(){console.log('completed');},
)

output: value: 4 + completed
```

Listato 2.19: AsyncSubject in RxJs

Molto simile al Replay e al Behaviour subject, memorizza solo l'ultimo valore emesso e lo notificherà agli osservatori solo se la sequenza è già terminata (figura 2.9). Utilizzato, ad esempio, con un hot Observable che potrebbe terminare prima di qualsiasi sottoscrizione. In questo caso, AsyncSubject può fornire ai suoi observer l'ultimo elemento emesso.

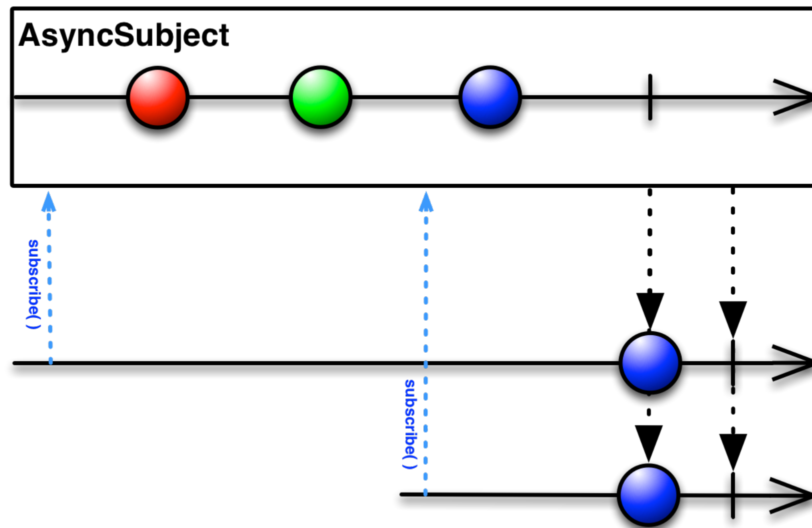

Figura 2.9: AsyncSubject

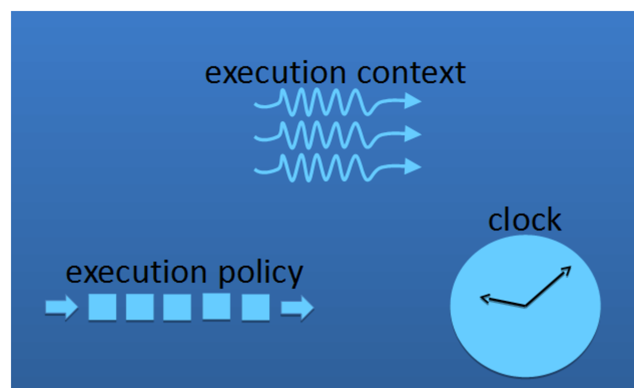

Figura 2.10: La natura di uno scheduler

Scheduling e concorrenza Lo scheduler (figura 2.10) stabilisce quando e come determinate operazioni devono avvenire quali, ad esempio, la sottoscrizione di una sorgente, invio della notifica agli ascoltatori. In particolare, consiste di tre elementi. Per prima cosa è una struttura dati: ogni operazione

da completare viene inserita nello scheduler e accodata insieme alle altre in base alla sua priorità o ad altri criteri. Esso offre un contesto di esecuzione che indica dove e quando l'operazione deve avvenire (immediatamente, nel thread corrente, in un meccanismo di callback). Come ultima cosa, ha un orologio che fornisce una nozione del tempo, diversa da quella usata nella vita di tutti i giorni; ogni operazione assegnata ad un particolare scheduler aderirà al tempo indicato dall'orologio di questo.

RxJS prevede diversi tipi di scheduler ciascuno dei quali può essere creato e ottenuto chiamando proprietà statiche sull'oggetto scheduler.

- ImmediateScheduler (ottenuto attraverso la proprietà statica immediate) inizierà l'esecuzione dell'operatore specificato immediatamente quindi in modo sincrono.

- CurrentThreadScheduler (proprietà currentThread) l'operatore verrà eseguito sullo stesso thread che lo ha invocato ma non immediatamente, verrà messo in una coda ed eseguito non appena l'operatore corrente ha terminato la sua esecuzione.

- DefaultScheduler (proprietà default) l'operatore verrà eseguito in una callback asincrona.

- ...

Gli operatori degli Observable abilitati ad operare in concorrenza possono ricevere in ingresso un parametro opzionale, lo scheduler appunto. Se non viene definito esplicitamente uno scheduler RxJS ne sceglierà uno di default utilizzando il principio della "minima concorrenza". Ad esempio, per gli operatori che restituiscono una sequenza con un numero finito e piccolo di messaggi, RxJS utilizzerà ImmediateScheduler; per operatori che restituiscono un observable con un numero potenzialmente elevato o infinito di elementi verrà invocato CurrentThreadScheduler, per gli operatori che utilizzano il tempo il default scheduler.

Attraverso il metodo observeOn è possibile specificare il contesto nel quale vogliamo sia chiamato il metodo onNext dell'observer.

```
Rx.Observable.generate(
    0,
    function () { return true; },
    function (x) { return x + 1; },
    function (x) { return x; }
    )
    .observeOn(Rx.Scheduler.default)
    .subscribe(...);
```

Listato 2.20: Scheduler

Alternativamente lo scheduler può essere passato direttamente al metodo generate().

Se si ha una catena di operatori è bene richiamare il metodo observeOn solo alla fine, poiché si presume che il numero di elementi della sequenza finale sia minore di quella iniziale; se specificassimo il contesto di esecuzione (lo scheduler) anche per gli operatori intermedi alla catena, c'è il rischio di fare lavoro inutile poiché, per ogni elemento emesso da questo e, magari non selezionato per la sequenza finale, viene richiamato lo scheduler indicato.

In implementazioni server-side di Rx lo scheduler gioca un ruolo importante poiché permette di assegnare lavori pesanti ad un pool di thread o ad un thread dedicato ed inviare il risultato di tale lavoro (invocazione del metodo onNext) ad un altro thread, ad esempio, al thread dell'interfaccia grafica in modo che l'utente venga notificato immediatamente del completamento dell'operazione.

Quando si usa RxJS difficilmente occorre preoccuparsi del giusto scheduler da passare ai vari operatori; Javascript è un linguaggio single-threaded, non ci sono molto opzioni di scheduling e, la maggior parte delle volte, il default scheduler è la scelta più appropriata.

Test Testare codice asincrono non è affatto cosa facile, fortunatamente l'estensione reattiva per Javascript permette di farlo in modo semplice. Se si dispone di una sequenza osservabile che emette valori in un periodo prolungato nel tempo, testarla in tempo reale diventa arduo. A tale scopo la libreria RxJS fornisce TestScheduler che permette di testare tale codice senza aspettare in realtà il passare del tempo. TestScheduler eredita da VirtualScheduler e permette di creare, pubblicare, sottoscrivere sequenza in un momento arbitrario e fittizio. Per esempio, possiamo mappare una sequenza che impiega cinque giorni per emettere i suoi elementi in una che impiega solo due minuti: è come se tutto venisse riportato in una scala ridotta. I metodi startWithTiming, startWithCreate e startWithDispose permettono di eseguire tutte le operazioni pianificate finché la coda che le mantiene è vuota, oppure è possibile specificare un tempo in modo che vengano eseguite solo le operazione che si riescono a completare nel tempo indicato.

L'esempio che segue (codice 2.21) mostra un hot observable che specifica le sue notifiche onNext, successivamente inizia testScheduler il quale specifica il momento in cui avviene la sottoscrizione (subscibe) e il rilascio (dispose) della sequenza.

Dopo il completamento della sequenza usiamo il metodo assertEqual di collectionAssert e andiamo a verificare se gli elementi emessi dalla sequenza sono

uguali a quelli che ci aspettiamo (stesso numero di elementi, ordine corretto . . .). Nel nostro esempio, dal momento che la sottoscrizione ha inizio a 150, il valore 'abc' mancherà. Possiamo notare che i valori emessi fino a 400 sono quelli emessi dopo la sottoscrizione e che la notifica onComplete avviene a 500 come specificato.

Per effetture test in RxJs esistono diversi framework come QUnit, Mocha, Jasmine ecc, questo esempio è stato realizzato con QUnit.

```
function createMessage(expected, actual) {
  return 'Expected: [' + expected.toString() + ']\r\nActual: [' +
      actual.toString() + ']';
}

// Using QUnit testing for assertions
var collectionAssert = {
  assertEqual: function (actual, expected) {
    var comparer = Rx.internals.isEqual, isOk = true;

    if (expected.length !== actual.length) {
      ok(false, 'Not equal length. Expected: ' + expected.length + '
          Actual: ' + actual.length);
      return;
    }

    for(var i = 0, len = expected.length; i < len; i++) {
      isOk = comparer(expected[i], actual[i]);
      if (!isOk) {
        break;
      }
    }

    ok(isOk, createMessage(expected, actual));
  }
};

var onNext = Rx.ReactiveTest.onNext,
  onCompleted = Rx.ReactiveTest.onCompleted,
  subscribe = Rx.ReactiveTest.subscribe;

test('buffer should join strings', function () {
  var scheduler = new Rx.TestScheduler();

  var input = scheduler.createHotObservable(
    onNext(100, 'abc'),
```

```
    onNext(200, 'def'),
    onNext(250, 'ghi'),
    onNext(300, 'pqr'),
    onNext(450, 'xyz'),
    onCompleted(500)
  );

  var results = scheduler.startScheduler(
    function () {
      return input.buffer(function () {
        return input.debounce(100, scheduler);
      })
      .map(function (b) {
        return b.join(',');
      });
    },
    {
      created: 50,
      subscribed: 150,
      disposed: 600
    }
  );

  collectionAssert.assertEqual(results.messages, [
    onNext(400, 'def,ghi,pqr'),
    onNext(500, 'xyz'),
    onCompleted(500)
  ]);

  collectionAssert.assertEqual(input.subscriptions, [
    subscribe(150, 500),
    subscribe(150, 400),
    subscribe(400, 500)
  ]);
});
```

Listato 2.21: Test in RxJS

2.3 RxPHP

Recentemente è stata sviluppata anche l'estensione reattiva per php (RxPHP), la versione corrente è la 1.1.0.

Le caratteristiche di RxJS le si possono ritrovare anche nell'estensione reattiva di PHP, non verranno perciò, ripresi nel dettaglio i vari elementi che lo contraddistinguono.

Osservabili e osservatori Di seguito, viene mostrato come sia possibile creare un observable da una array e come avviene la sottoscrizione di questo da parte di un observer (2.22).

```
$source = \Rx\Observable::fromArray([1, 2, 3, 4]);

$subscription = $source->subscribe(new \Rx\Observer\CallbackObserver(
    function ($x) {
        echo 'Next: ', $x, PHP_EOL;
    },
    function (Exception $ex) {
        echo 'Error: ', $ex->getMessage(), PHP_EOL;
    },
    function () {
        echo 'Completed', PHP_EOL;
    }
));

Output:
Next: 1
Next: 2
Next: 3
Next: 4
Completed
```

Listato 2.22: creazione di un observable e un observer in RxPHP

Operatori in catena Viene riportato un esempio (listato 2.23) che mostra come sia possibile applicare operatori in catena da Observable in php.

```
$source1 = \Rx\Observable::just(42);
$source2 = \Rx\Observable::just(56);
$source = (new
    \Rx\Observable\EmptyObservable())->concat($source1)->concat($source2);
$subscription = $source->subscribe($stdoutObserver);

Next value: 42
Next value: 56
Complete!
```

Listato 2.23: Operatore concat in RxPHP

Capitolo 3

Caso di studio: il progetto AvvocaTimer

In questo capitolo non verrà riportata la progettazione e implementazione dell'intero sistema AvvocaTimer ma, verranno selezionate una parte della web application è una piccola parte di backend e rivisitate con il nuovo approccio della programmazione reattiva. Si procede dunque, con la progettazione, implementazione e collaudo di tali moduli. Come si potrà notare la parte di front-end è molto più corposa rispetto a quella di back-end poiché maggiore è la necessità di gestire grandi flussi asincroni di eventi e dati.

3.1 Introduzione

AvvocaTimer è un sistema di time tracking per studi legali che permette ad avvocati di mantenere traccia delle ore lavorate ai diversi casi e delle attività svolte sugli stessi. In particolare, il sistema si occupa delle gestione di più studi legali ciascuno dei quali, con un proprio "staff" di avvocati e una serie di casi ai quali sta lavorando. Ad ogni caso possono essere associati più faldoni ciascuno contenente parte della documentazione di questo; inoltre, ciascun faldone è provvisto di un tag NFC e un QR code che permette di identificarlo univocamente all'interno del sistema. Quando un avvocato, utente del sistema correttamente registrato, intende iniziare a lavorare ad un determinato caso cioè, su uno specifico faldone, dovrà attivare un timer sullo stesso e stopparlo solo a lavoro ultimato. L'identificazione del faldone sul quale attivare il timer avviene attraverso la lettura del tag NFC o del QR code posto su questo. Come già accennato, il sistema si occupa della gestione di più studi legali ciascuno con più avvocati: per entrare a far parte di un determinato ufficio un avvocato deve ricevere un invito e, solo successivamente potrà registrarsi,

alternativamente, può creare un proprio studio diventandone amministratore. I clienti sono condivisi da tutti gli uffici.

Il sistema è composto da un'applicazione mobile ibrida e una web application. L'applicazione mobile permette la lettura del tag NFC e del QR code, lo start e/o stop del timer sui diversi faldoni, la gestione dei clienti, dei casi e dei faldoni delle studio ai quali l'utente appartiene ecc...
La web app si occupa di mostrare informazioni di sintesi sul lavoro svolto dall'avvocato, ad esempio, potrà visualizzare le ore lavorate ad un determinato caso oppure, le ore lavorate ai casi di un determinato cliente. Solo l'utente amministratore ha la possibilità di visualizzare report sul lavoro svolto dagli altri utenti appartenenti al proprio studio.

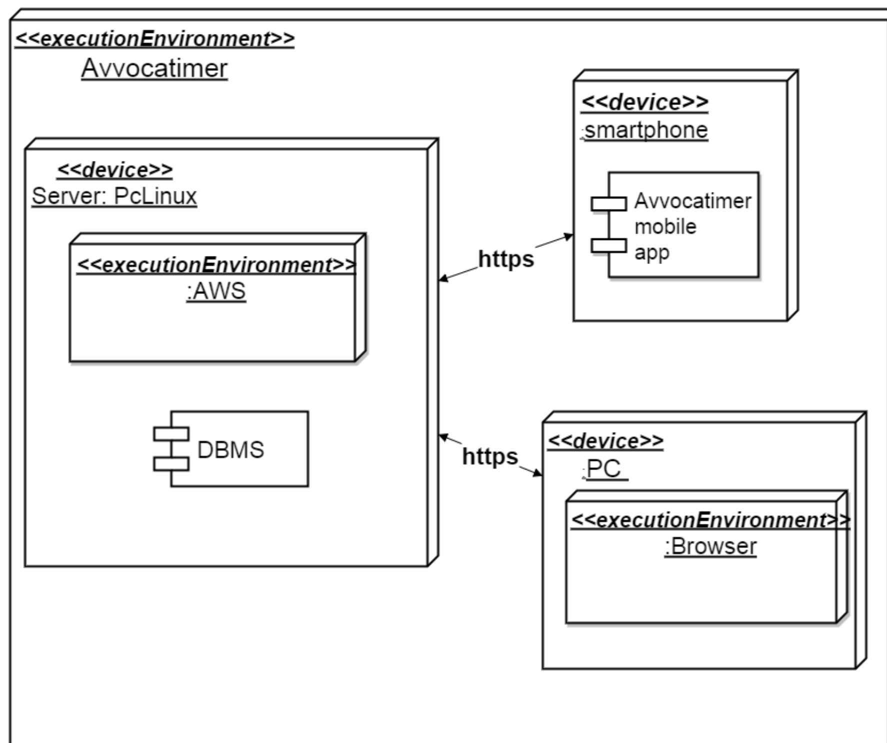

Figura 3.1: Diagramma di deployment del sistema

Il progetto è stato realizzato in collaborazione con due ragazzi del corso di laurea: ognuno con assegnata una determinata parte. Personalmente mi sono occupata dello sviluppo di api per l'applicazione mobile e del front-end della web application. Parte del front-end e back-end del progetto Avvocatimer è stato nuovamente progettato e sviluppato con il paradigma di programmazione reattivo; di seguito, verrà descritto come sia stato potuto apportarlo a tale caso di studio e quali siano i vantaggi derivanti del suo utilizzo.

3.2 Analisi dei requisiti

Come già accennato, il progetto è stata svolto in collaborazione con altri ragazzi, personalmente mi sono occupata dello sviluppo della web application e delle api per i client.

La web application deve prevedere quasi la totalità delle funzioni offerte dall'applicazione mobile e non solo; nel dettaglio e, come si può osservare dal diagramma dei casi d'uso (figura 3.2), deve permettere all'utente di:

- registrarsi al sistema e creare un proprio studio.

- Effettuare la login.

- Visualizzare l'elenco dei lavori effettuati.

- Visualizzare e inserire nuovi clienti, casi e faldoni.

- Visualizzare grafici che sintetizzano il lavoro effettuato ai vari casi, faldoni, per un cliente o in base all'attività svolta.

- Modificare la nota o l'attività associata ad un determinato lavoro.

- Stoppare il timer attivo su uno specifico faldone.

- Visualizzare l'elenco degli utenti appartenenti al proprio ufficio e invitarne altri.

- Visualizzare informazioni di sintesi sul lavoro svolto dai membri del proprio ufficio (solo per utente amministratore).

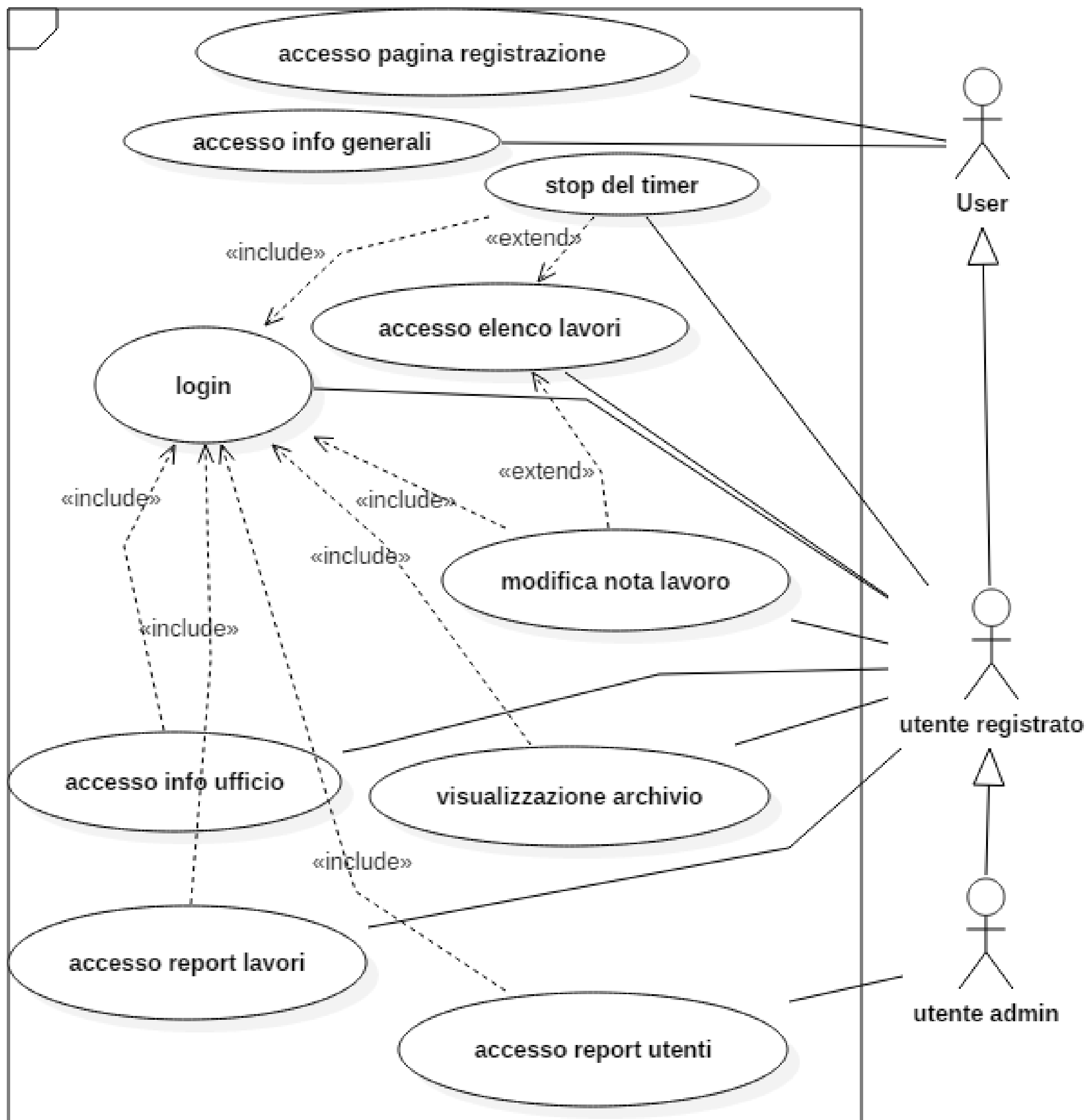

Figura 3.2: Diagramma dei casi d'uso della web application

Non tutta l'applicazione è stata modificata e integrata con il paradigma di programmazione reattiva per questo sono stati selezionati determinati moduli e, in particolare, quelli che si occupano di:

- mostrare all'utente gli ultimi cinquanta lavori effettuati.

- creare, per ogni giorno della settimana corrente, un grafico che mostra quanto l'utente ha lavorato ai vari faldoni.
- permettere di modificare l'attività svolta e/o la nota di un determinato lavoro.
- aggiornare la pagina alla pressione del corrispondente bottone.
- mostrare un grafico a torta che evidenzia il tempo lavorato ai vari faldoni di uno specifico caso dell'ufficio.

La parte di progetto a me assegnata prevedeva anche lo sviluppo di api per i client (gestione delle richieste e relative risposte), si è voluto introdurre e testare il paradigma reattivo anche nella parte back-end del sistema. La RP è stata utilizza nel back-end per gestire le richieste dei lavori dell'utente.

3.3 Progettazione

Attraverso la Reactive programming verranno progettate le parti sopra individuate e, in, particolare, verrà utilizzata per gestire eventi, richieste e/o risposte e animazioni dell'interfaccia grafica.

3.3.1 Parte front-end

Quando l'utente accede alla home dell'applicazione o quando preme il pulsante "aggiorna", si vogliono mostrare tutti i lavori da lui effettuati all'interno di una tabella. Decido di rappresentare la pressione del pulsante "aggiorna" come uno stream, ogni elemento del quale rappresenta tale evento.

———x——x——x—— `clickRefreshButtonStream`

Lo schema precedente rappresenta un observable creato dall'evento "pressione del bottone": la x raffigura l'evento mentre, la riga orizzontale, il tempo che scorre da sinistra verso destra.

Quando viene premuto il pulsante è necessario effettuare una richiesta ajax al server per recuperare tutti i lavori effettuati dell'utente; una volta ottenuta la risposta si procede ad analizzarla e con la creazione della relativa tabella.

———x——x——x—— `clickRefreshButtonStream`
map(x in r)
———r——r——r—— `requestStream`
flatMap(r in –a–)

———a—a—a– `responseStream`

Ogni elemento della seconda e terza sequenza rappresentano rispettivamente una richiesta e una risposta al server; quest'ultima può essere in qualsiasi formato. Come si può vedere dallo schema, ogni elemento emesso dal primo stream viene trasformato, attraverso la funzione map, in un evento di richiesta di informazioni al server, a sua volta, ogni richiesta viene convertita in un evento di risposta e, in particolare, l'operatore flatMap la converto in uno nuovo observable. La risposta non avviene contestualmente alla richiesta ma, in modo asincrono, anche in un tempo successivo.

`ResponseStream` è stato definito in accordo con `requestStream`, se avvengono uno o più eventi nella prima sequenza si avranno i corrispondenti eventi di risposta nella seconda.

Poiché i dati dovranno essere visualizzati anche al caricamento della pagina occorre far emettere allo stream di eventi "pressione del bottone", un primo elemento prima che inizi con la sua sequenza. In questo modo, anche al caricamento della pagina sarà effettuata una richiesta di dati al server e l'utente potrà visualizzare i proprio lavori senza necessariamente premere il relativo bottone.

——x—x—x——- `clickRefreshButtonStream`
startWith(x)
–x—x—x—x——- `clickRefreshButtonStream`
map(x in r)
–r—r—r—r——- `requestStream`
flatMap(r in –a–)
—a—a—a—a—- `responseStream`

Ogni risposta della sequenza `responseStream` deve essere utilizzata per la creazione della tabella; supponendo che ogni risposta sia un array di lavori, per ognuno occorre creare una riga all'interno della tabella.

—-a———a——— `responseStream`
flatMap(r in –l–l—)
—-l-l-l———l-l-l—- `workStream`
map(l in t)
—-t-t-t——t-t-t—— `rowTableStream`

Come si può notare dallo schema, ogni risposta ottenuta dal server viene convertita, mediante l'operatore flatMap, in una nuova sequenza contente i lavori effettuati dall'utente mentre, l'operatore immediatamente dopo nella

catena, map, procede a mappare ogni lavoro in una riga della tabella. Compito dell'observer, sottoscrittore di tale sequenza, è quello di inserire ogni riga emessa nell'appropriato spazio.

Per evitare di riempire la tabella con i lavori ottenuti da tutte le richieste e quindi, righe ripetute, occorre svuotare la tabella dopo ogni richiesta e, ripopolarla quando si ottiene la corrispondente risposta. Andrò,dunque a sottoscrivere la sequenza `requestStream` in modo che, per ogni elemento da questa emesso, avrà luogo una 'ripulitura' della tabella dei lavori.

La pagina iniziale della web application dovrà, inoltre, visualizzare dei grafici a torta che mostrano quanto l'utente ha lavorato ai vari faldoni nei vari giorni della settimana corrente. Non sarà necessario effettuare ulteriori richieste al server poiché, ogni lavoro ottenuto dalle varie risposte contiene un timestamp di inizio lavoro. Con tale informazione, sarà possibile selezionare i lavori della settimana e, con questi, creare il dataset di dati necessario per creare i grafici.

—–a———a——— `responseStream`
flatMap(a in -l—l—)
—–l-l-l——-l-l-l—– `workStream`
filter(lavori della settimana corrente e conclusi)
—–l-l———l-l——- `filterWorkStream`
reduce(dataSet per grafici)
———-w————w—

Una volta ottenuta la risposta dal server e trasformata in una sequenza di lavori questi, vengono forniti in input all'operatore filter che si occupa di selezionare solo quelli il cui timestamp di inizio riporta la data di uno dei giorni della settimana corrente e, che è già terminato. L'ultimo operatore, reduce, è un operatore di aggregazione e quindi riporta come output un singolo valore dopo che la sequenza in input è terminata, in questo caso si occupa della creazione del dataset necessario a creare i grafici partendo dai lavori emessi dalla sequenza `filterWorkStream`. Il sottoscrittore di tale sequenza, una volta ricevute le informazioni necessarie alla creazione dei grafici, procede con la loro rappresentazione. Come per la tabella dei lavori, per non incorrere in errori dovuti alla duplicazione di informazioni, è necessario resettare la tabella dei grafici dopo ogni richiesta al server.

L'utente ha anche la possibilità di modificare l'attività effettuata e di inserire una nota per un determinato lavoro, in particolare, quando clicca su una riga della tabella dei lavori, apparirà una schermata che permette di apportare tali modifiche. Rappresento gli eventi "click sulla tabella" come uno stream, alla stessa maniera del "click sul bottone".

—–c—–c—–c—–c—- `clickTableStream`
filter(elemento TD)
—–c—–c—–c———- `filterClickTableStream`
map(riga tabella di c)
—–d—–d—–b———- `selectedRowTableStream`
untilChange()
—–d————b———- `selectedRowTableStream`

Il primo stream emette un nuovo elemento ogni qual volta si verifica l'evento "click sulla tabella", questo, viene fornito ad una funzione filter che seleziona solo gli eventi "click su un elemento td della tabella". Proseguendo nella catena, si hanno gli operatori map e untilChange che si occupano rispettivamente di trasformare l'evento nella riga della tabella in cui questo è avvenuto e di emettere la riga oggetto dell'evento solo se questa è diversa da quella emessa in precedenza: click ripetuti sulla stessa riga verranno ignorati.

L'observer di questa sequenza (`selectedRowTableStream`) viene notificato ogni volta che l'utente clicca su una riga diversa della tabella e si occupa di mostrare la finestra con dettaglio del lavoro selezionato. Tale schermata prevede tre diversi bottone che permetto all'utente di chiuderla, salvare le modifiche al lavoro e stoppare il timer che mantiene il conteggio dei minuti trascorsi rispetto all'orario riportato nel timestamp di inizio; quest'ultimo abilitato solo nel caso in cui il lavoro deve ancora terminare. Ogni volta che l'utente preme il pulsante per salvare le modifiche sul lavoro corrente, occorre effettuare una richiesta ajax al server passandogli l'attività e la nota, solo quando si ottiene una risposta positiva è necessario aggiornare le due tabelle della pagina.

Gli eventi "click sul bottone di modifica" vengono modellati come una stream al quale applichiamo operatori in catena.

—–c—–c—–c—–c———— `clickSaveChangesButtonStream`
map(c in d)
—–d—–d—–d—–d————– `stringRequestStream`
map(d in b)
—–b—–b—–b—–b———— `requestChangesStream`
flatMap(b in –a–)
———a———a———a———a `responseChangesStream`

Lo schema precedente mostra come ogni click del bottone di modifica viene trasformato in una stringa, corrispondente ad un url, che a sua volta, viene mappato in una richiesta ajax al server. Attraverso l'operatore flatMap da ogni risposta si ottiene un nuova sequenza.

Una volta che dal server arriva la conferma che le modifiche sono state apportate correttamente occorre aggiornare nuovamente l'intera pagina cioè fare le stesse operazione eseguite in precedenza con l'evento "click sul bottone aggiorna". Per questo motivo, gli stream `responseChangesStream` e `clickRefreshButtonStream` vengono uniti: ad ogni emissione di un valore da parte di uno dei due andrà fatta una nuova richiesta.

—–c—–c—–c—–c—–c— `responseChangesStream`
—a—–a—–a—–a—–a—- `clickRefreshButtonStream`
merge()
—a-c—a-c—a-c—a-c—a-c— `refreshStream`

Per facilitare l'utente nella lettura dei suoi lavori, ogni volta che viene selezionato una settore del grafo si vogliono evidenziare le righe della tabella utilizzate per realizzarlo. Ogni settore rappresenta il tempo di lavoro eseguito in un determinata giorno della settimana su un faldone, ci potranno essere più lavori effettuati lo stesso giorno e sullo stesso faldone dall'utente. Il valore dell'attributo id di ogni riga della tabella corrisponde all'identificativo del lavoro che ospita mentre, ogni settore del grafico mantiene l'array di id dei lavori a lui associati.

—o—–o—–o—–o—–o—– `mouseOverStream`
—–p—–p—–p—–p—–p— `mouseOutStream`
merge()
—o-p—o-p—o-p—o-p—o-p— `mouseMoveStream`
map(in d)
—d-d—d-d—d-d—d-d—d-d— `mouseMoveStream`

Il primo stream è un stream di eventi "mouse sopra un settore del grafico" mentre il secondo di eventi "mouse fuori dal settore del grafico". I due stream, attraverso l'operatore merge vengono uniti per formarne uno unico, ogni elemento di questo viene passato all'operatore della catena map che procede a trasformarlo in un array contenente due valori: un booleano, settato a vero quando è un evento 'mouseOver' e a falso alternativamente, e il settore del grafico. Ogni elemento emesso dalla sequenza `mouseMoveStream` contiene sia l'informazione circa il settore del grafo in cui l'evento è avvenuto sia il tipo di evento. L'observer di tale sequenza andrà a evidenziare le righe della tabella corrispondi al settore del grafico quando il valore del booleano emesso è uguale a true ad eliminare la sottolineatura altrimenti.

Si procede ora con la progettazione della parte della applicazione che mostra

all'utente un grafico a torta creato secondo le ore lavorate ai diversi faldoni di un determinato caso: per prima cosa l'utente deve selezionare il caso poi avverrà la creazione del grafico. Decido di rappresentare l'evento 'selezione di un caso' con uno stream:

——s————s——————— `selectedCaseChangeStream`

Ad ogni istanza di tale evento deve corrispondere una richiesta al server per ottenere i dati necessari alla creazione del grafico.

——s————s——————— `selectedCaseChangeStream`
map(s in d)
——d————d——————— `requestDataChartDossierStream`
flatMap(d in –a–)
———a—————a——————— `responseDataChartDossierStream`

Una volta ottenuta la risposta con i dati che ci occorrono viene creato uno stream dai sui elementi (si suppone che la risposta sia un array di array), con l'operatore map selezionate solo determinate informazioni, infine, attraverso l'operatore scan, viene creato il dataset del grafico. L'observer non farò altro che passare tale dataset alla funzione incaricata della creazione del grafico.

——-b-b-b-b-b—————— `recordChartDossierStream`
map(b in p)
——-p-p-p-p-p——————- `recordChartDossierStream`
scan()
————————z———- `datasetChartDossier`

3.3.2 Parte back-end

Quando al server vengono richiesti i lavori effettuati da un determinato utente, esso dovrà interrogare il database e restituire solo gli ultimi 50 lavori. Come si può notare dalla schema che segue, la richiesta viene modellato come uno stream, a sua volta convertita in un'interrogazione al database la quale, restituirà una array contenete tutti i lavori dell'utente. Occorre quindi selezionare da ciascun lavoro solo le informazioni di cui il client necessita e prendere gli ultimi 50 elementi (i lavori all'interno dell'array sono ordinati secondo il valore di timestamp e in ordine decrescente cioé dal lavoro più recente a quello più datato)

——r———————————— requestStream
map(r in d)
——d———————————— requestDbStream
flatMap(d in –p–)
——p-p-p-p-p-p-p———————— workStream
map(p in b)
——b-b-b-b-b-b-b———————— workStream
take(50)
——b-b-b-b-b-b-b———————— workStream
toArray()
——[b-b-b-b-b-b-b]———————— works

3.4 Implementazione

Si procederà ora con l'implementazione della parte progettata nella precedente sezione utilizzando le estensioni reattive dei linguaggi adottati per lo sviluppo del progetto: RxJS e RxPHP.

3.4.1 Parte front-end

Per l'implementazione in chiave reattiva è stata utilizzata, oltre alla libreria RxJS, RxJS-DOM, una libreria creata dallo stesso team di RxJS, la quale prevede operatori utili per maneggiare facilmente il DOM e eventi correlati al browser. Inoltre, sono state utilizzate le seguenti tecnologie-librerie: HTML, css, javascript, jQuery, bootstrap.

Tutto il codice che segue deve essere eseguito solo dopo che il DOM è pronto poiché negli script verranno utilizzati parte dei suo elementi: si vuole caricare il codice dopo che è avvenuto l'evento DOMContentLoaded perché rappresentata il momento in cui nel browser sono presenti tutti gli elementi delle pagina. RxJS-DOM prevede l'Observable Rx.DOM.ready() il quale emette un elemento proprio nel momento in cui si verifica l'evento DOMContentLoaded, precedo, quindi, con l'inserimento del codice in una funzione `initialize()` che verrà utilizzata per sottoscrivere Rx.DOM.ready() (listato 3.1).

```
function initialize(){
   //manipolazione elementi DOM
}

Rx.DOM.ready().subscribe(initialize);
```

Listato 3.1: Rx.DOM.ready() Observable

Come mostrato in (listato 3.2) gli eventi di pressione dei bottoni per aggiornare la pagina e per effettuare modifiche ad un determinato lavoro sono gestiti come due observable i quali, vengono uniti attraverso la funzione merge() poiché ad ogni evento emesso da una delle due sequenza occorre effettuare le stesse operazioni. L'operatore debounce() emette un elemento dall'observable dopo che è trascorso un determinato periodo di tempo, in questo caso se si verifica più di un evento nell'arco di 500 millisecondi, questi verranno considerati come un singolo evento, ciò permette di evitare continui e inutili aggiornamenti della pagina nel breve periodo.

```
function initialize(){

   var btnSaveChanges = $('#btn_saveChanges');
    var pathcontrollerSaveChanges =
        Routing.generate('saveNoteAndWork').toString();

    var saveChangesClickStream =
        Rx.Observable.fromEvent(btnSaveChanges, 'click')
        .map(function(){
            var s =
                "officeWorkId="+getOfficeWorkId()+"&note="+getNote()
                +"&workId="+getIdWork();
            console.log(s);
            return s;
        })
        .flatMap(function(s){
            return Rx.DOM.ajax({ url:
                pathcontrollerSaveChanges+"?"+s, responseType: 'json'})
        });

   var btnRefresh = $('#btn_refresh');
    var refreshStream = Rx.Observable.fromEvent(btnRefresh, 'click')
        .merge(saveChangesClickStream)
        .debounce(500);
}
```

Listato 3.2: refreshStream

Per ogni elemento emesso da refreshStream, observable ottenuto in precedenza unendo due sequenze, occorre effettuare una richiesta al server per ottenere i lavori dell'utente e ripulire la relativa tabella. Come si può notare

dal codice 3.3 ogni evento emesso dall'observable di partenza, attraverso l'operatore flatMap, viene trasformato in una una sequenza che corrisponde alla risposta pervenuta dal sever.

L'operatore finale retry(), in caso di errore, non lo propaga immediatamente all'observer di tale sequenza ma, prova ad effettuare tre richieste al server, solo nel caso in cui tutte falliscono l'errore viene riportato: questo può essere utile, ad esempio, quando si verificano momentanei problemi di connessione e si vuole gestire l'erore direttamente all'interno della sequenza.

```
refreshStream
   .subscribe(function(){
        $("#work_table tbody tr").remove();
    });

var urlWork = Routing.generate('rxWeekWork').toString();
var responseStream = refreshStream
   .startWith('startup click')
   .flatMap(function(){
     return Rx.DOM.ajax({ url: urlWork, responseType: 'json'})
   })
   .retry(3);
```

Listato 3.3: responseStream

Nella porzione di codice (listato 3.4), la risposta del server, essendo un array, viene trasformata in una nuova sequenza: ogni lavoro, attraverso l'operatore map e, all'ausilio della funzione makeRow(work), sarà convertito in un elemento tr del DOM.

Per evitare di modificare il DOM ogni volta che viene emesso un nuovo lavoro e quindi ridisegnare inutilmente la pagina più volte, viene utilizzato bufferWithTime che si occupa di bufferizzare tutte le righe emesse dal precedente operatore, negli ultimi 500 millisecondi e di passarle come array all'operatore successivo (array vuoto se non è stato emesso alcun valore).

Per fare questo senza la programmazione reattiva sarebbe stato necessario mantenere un bufffer per le varie righe e un conteggio da resettare dopo ogni periodo, ma grazie all'operatore bufferWithTime, tutte questo viene fatto automaticamente.

Successivamente le varie righe vengono inserite in un fragment ma, siccome non è all'interno del DOM, modificare il suo contenuto è efficiente e veloce.

Come ultima cosa il fragment andrà inserito all'interno del DOM: si avrà una sola modifica della pagina per aggiungere più righe della tabella dei lavori.

```
var rowStream = responseStream
```

```
    .flatMap(function(data){
      return Rx.Observable.from(data.response);
    })
    .map(makeRow)
    .bufferWithTime(500)
    .filter(function(rows){return rows.length > 0})
    .map(function(rows){
      var fragment = document.createDocumentFragment();
      rows.forEach(function(row){
        fragment.appendChild(row)
      });
      return fragment;
    });

rowStream
    .subscribe(
      function(fragment){
        table.appendChild(fragment);
      },
      function(error){
        console.log(error);
      },
      function(){
        console.log('row stream completed!');
      }
    );

//funzione che prende in ingresso un lavoro e crea un elemento tr a
     partire da questo
function makeRow(work){
    var row = document.createElement('tr');
    row.id = work.id;
    [work.id, work.idDossier, work.nameDossier, work.name,
        work.workName, work.hoursOfWork, work.hoursOfWork
            ].forEach(function (text) {
        var cell = document.createElement('td');
        cell.textContent = text;
        row.appendChild(cell);
    });
    return row;
}
```

Listato 3.4: rowStream

La risposta del server viene utilizzata sia per la creazione della tabella dei lavori sia per la realizzazione dei grafici. Come si può osservare dal frammento di codice (listato 3.5) dalla risposta si ottiene una sequenza di lavori, da questa vengono filtrati solo quelli il cui timestamp di inizio riporta la data di un giorno della settimana corrente e passati all'operato di aggregazione reduce. Tale operatore permette di creare una array di sette array, uno per ogni giorno della settimana, ciascuno contente parte delle informazioni dei lavori effettuati in quel determinato giorno. Ad esempio, workArray[0] è un array contente tanti elementi quanti sono stati i lavori effettuati dall'utente il lunedì, ogni elemento, a sua volta, è un array contenente la durata, l'id e il faldone oggetto del lavoro.

L'observer di tale sequenza, una volta ottenuto il dataset di ogni grafico precede con la sua rappresentazione.

```
var dataChartsStream = responseStream
  .subscribe(
    function(x){
      return Rx.Observable.from(x.response)
        .filter(function(work){
          var dataTimestamp =
              getDataTimestamp(work.startTimeStamp);
          return ((dataTimestamp.getTime() >=
              getMonday.getTime()) && (dataTimestamp.getTime() >=
              getSunday.getTime()));
        })
        .reduce(function(workArray, work){
          var secondOfWork = getSecondOfWork(work.hoursOfWork);
          var tableTD = getTDTableForWork(work.startTimeStamp);
          workArray[tableTD].push([work.nameDossier,
              secondOfWork, work.id]);
          return workArray;
        }, [[],[],[],[],[],[],[]])
        .subscribe(
          function (workForChartArray) {
            for (var i=0; i<workForChartArray.length; i++) {
              if (!(workForChartArray[i] == 0)) {
                drawChartWeekWorkTable('chartDay' + i,
                    workForChartArray[i]);
              }
            }
          },
          function (err) {
            console.log('error: ' +err);
```

```
                },
                function (){
                   console.log('dataChart stream Completed! ');
                }
             )
      }
    );
```

Listato 3.5: dataChartsStream

La risposta proveniente dal sever è stata opportunamente elaborata e sono state prodotte le due tabelle, ora si può passare alla gestione degli eventi sulla tabella dei lavori, come già descritto nella fase di progettazione, ad ogni click su una cella si vuol mostrale all'utente una schermata per permettergli di modificare informazioni sul lavoro.

Sfruttando il fatto che gli eventi vengono propagati anche agli elementi genitori all'interno del DOM, invece di creare un observable per ogni riga della tabella che emette gli eventi che su questa si verificano, viene semplicemente create un sequenza di eventi 'click' sulla tabella.

La funzione getRowFromEvent(event) (codice 3.6) crea un observable dagli eventi 'click' sulla tabella, procede a verificare se si tratta di un evento su una cella (elemento td) e, attraverso l'operatore pluck procede a recupera il suo genitore nella gerarchia dei componenti della pagine, in questo caso, la riga che la ospita.

L'ultimo operatore della catena distinctUntilChanged() emette una nuova riga solo se questa è diversa da quella precedentemente emessa.

L'osservatore della sequenza mostra all'utente un modal con un campo per inserire la note e/o il lavoro effettuato.

```
function getRowFromEvent(event){
    var table = $('#work_table');
    return Rx.Observable.fromEvent(table, event)
        .filter(function(event){
            var el = event.target;
            return el.tagName === 'TD';
        })
        .pluck('target', 'parentNode')
        .distinctUntilChanged();
}

getRowFromEvent('click')
   .subscribe(function(row){
```

```
    var idWork = $(".modal-body #idWork
        ").text(row.getElementsByTagName('td')[0].innerHTML);
    var nameDosier = $(".modal-body
        #nameAndIdDossier").text(row.getElementsByTagName('td')[2].innerHTML);
    $('#myModal').modal();
});
```

Listato 3.6: selectedRowTableStream

Quando l'utente si posiziona con il mouse in un settore di un grafico si vogliono evidenziare le righe della tabella che contengono i lavori a lui correlati e quindi, utilizzati per la creazione. Come rappresentato in (listato 3.7), vengono generate due sequenza rispettivamente dagli eventi 'mouse over' e 'mouse out', successivamente unite e sottoscritte da un osservatore il cui compito è quello di evidenziare determinate righe della tabella ('mouse over') oppure decolorarle ('mouse out'). Anche in questo caso, non vengono creati tanti observable quanti sono i grafici o i settori rappresentati ma, viene creato un unico observable che gestisce tutti gli eventi 'mouse sopra e fuori' della tabella poiché, grazie alla caratteristica degli eventi di propagarsi fra gli elementi del DOM, ogni evento su un settore di un determinato grafico viene riportato all'elemento genitore della pagina e quindi, alla tabella.

```
var mouseMoveStream = Rx.Observable.fromEvent(bOver, 'custom')
   .map(function(e){
      return {x:true, id:e['idRow']}
   })
   .merge(
      Rx.Observable.fromEvent(bOut, 'custom')
         .map(function(e){
            return {x:false, id:e['idRow']}
         })
   );

mouseMoveStream
   .subscribe(
      function(bool) {
         //console.log('mouse:' +bool.x);
         $("#" + bool.id).css('background', bool.x ? '#FFFF00' :
             '#FFFFFF');
      }
   );
```

Listato 3.7: mouseMoveStream

Dalla home dell'applicazione è permesso all'utente navigare alla pagina che riporta i report del lavoro da lui effettui. In (codice 3.8) è riportato il frammento di codice che si occupa della creazione del grafico che mostra il lavoro effettuato ai vari faldoni di un determinato caso con la libreria RxJS. La creazione di tutti gli altri grafici è stata implementata in maniera analoga. Ogni volta che l'utente seleziona un nuovo caso dalla selected list dell'interfaccia grafica, viene fatta una richiesta al server per ottenere i dati necessari alla creazione del grafico. L'operatore `switchLatest` seleziona l'ultima risposta ricevuta e si procede con la decodifica dei dati giunti dal server che sono in formato json. Dopo tale decodifica si ottiene un array di array cioè il giusto dataset per la creazione del grafico.

```
var selectBox = $(document).find('#caseChartDossier');
var dataForDossierChart = Rx.Observable.fromEvent(selectBox,
    'onChange')
    .debounce(500)
    .map(function(e){
        return url =
            Routing.generate('getDataForDossierChart').toString() +
            e.getSelection.val();
    })
    .flatMap(function(url){
        return Rx.DOM.ajax({ url: url, responseType: 'json'});
    })
    .retry(3)
    .switchLatest()
    .map(function(data){
        return JSON.parse(data.response);
    });

dataForDossierChart.subscribe(
    function(dataChart){
        drawChart(dataChart);
    },
    function(error){
        console.log(error);
    });
```

Listato 3.8: Codice reattivo per la creazione del grafico dei faldoni.

Le immagini (figura 3.3) e (figura 3.4) raffigurano rispettivamente la pagina principale dell'applicazione con le tabelle dei lavori e dei grafici e, il modal che permette di modificare la nota e l'attività eseguita nello svolgimento di un particolare lavoro.

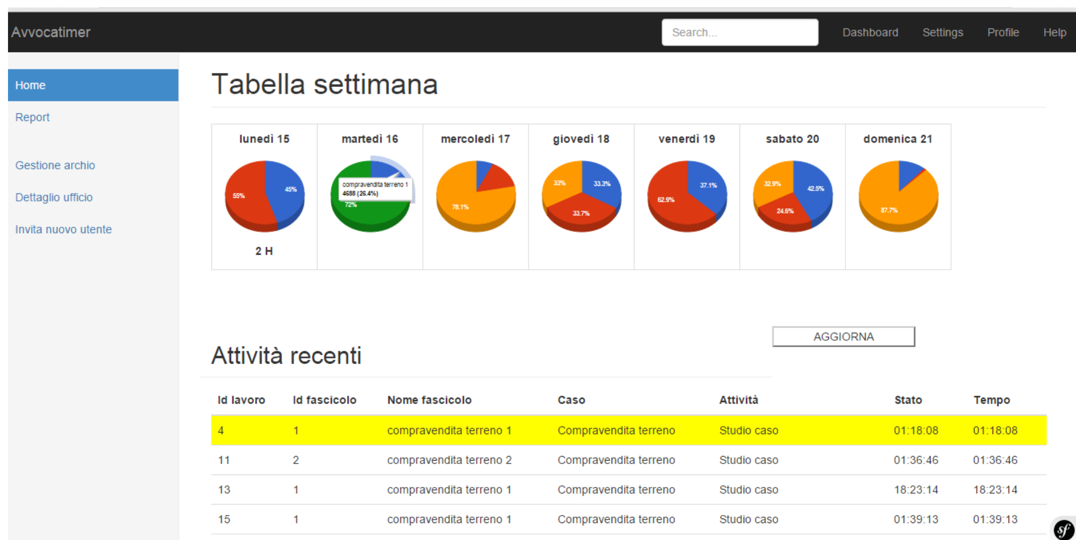

Figura 3.3: Pagina principale della web application.

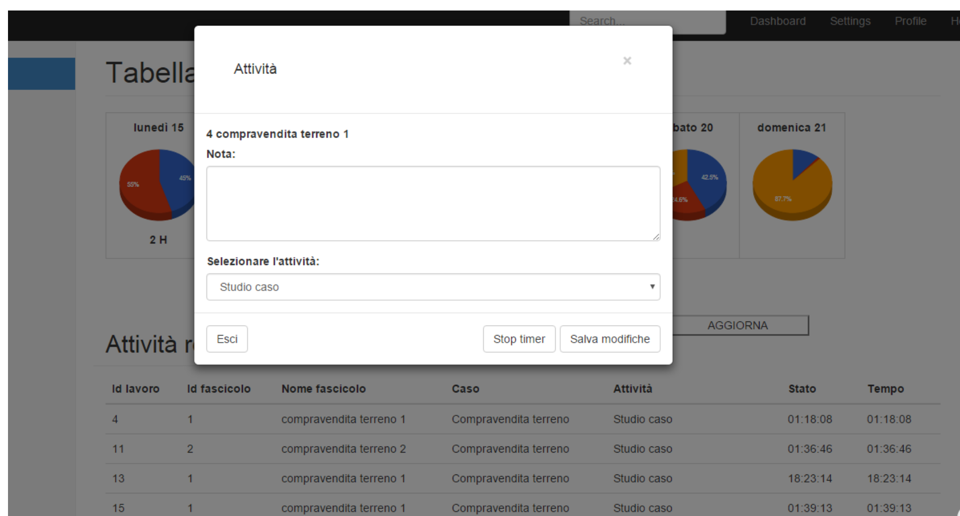

Figura 3.4: Avvocatimer con modal per inserimento e/o modifica dell'attività e della nota di un determinato lavoro.

La figura (figura 3.5) mostra la pagina della web application dei report, in particolare, mostra, nella prima riga, il grafico del lavoro effettuato ai diversi casi dello studio e ai diversi faldoni di uno specifico caso. Nella seconda riga, il primo grafico sintetizza il lavoro effettuato per il cliente selezionato mentre, il secondo le attività svolte su un faldone.

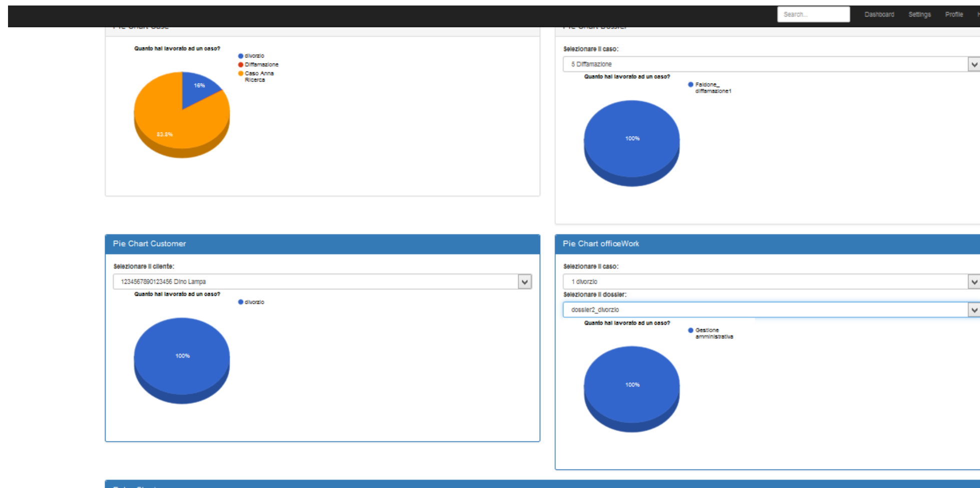

Figura 3.5: Pagina della web application dei report.

3.4.2 Parte back-end

La parte di back-end rivisitata in chiave reattiva si occupa di gestire le richieste del client per i lavori svolti dall'utente. Come mostrato in 3.9 il server una volta ricevuta la richiesta procede ad interrogare opportunamente il database recuperando i lavori dell'utente. Con questi, viene creata un sequenza, ogni elemento della quale modificato attraverso l'operatore map e selezionati solo i primi 50 elementi (operatore take).

```
$createStdoutObserver = function () {
   return new CallbackObserver(
      function ($value) { return new JsonResponse($value); },
      function ($error) { return new JsonResponse($error); },
      function ()       { return new JsonResponse('Completed!'); }
   );
};
$stdoutObserver = $createStdoutObserver();

//vengono recuperati i lavori dell'utente dal database
$mysqlManager = $this->get('mysql_manager');
$timeOfWorkOnCases = $mysqlManager->getUserWork(getCurrentUser());

//viene creato un observable di lavori e presi gli ultimi 50
Observable::fromArray($timeOfWorkOnCases)
   ->map(function ($work) {
      return array($work['officeWorkName'],
         intval($work['secondsOfWork']));
```

```
})
->take(50)
->toArray()
->subscribe($stdoutObserver);
```

Listato 3.9: porzione di codice php per gestire la richiesta del client dei lavori dell'utente

3.5 Collaudo

In questa sezione verrà riportato un test che mira a verificare la correttezza di una piccola parte del codice scritto con il paradigma reattivo.

Test con TestScheduler TestScheduler è uno scheduler (come descritto nel secondo capitolo) progettato per aiutare il programmatore nella creazione dei test perché permette l'esecuzione di azioni in un tempo 'virtuale'.

Nel frammento (codice 3.10) viene ripresa la parte della web application che si occupa della creazione di un nuovo observable dai lavori ottenuti dal server, di trasformarli in un elemento TD del DOM e di inserirli all'interno della corrispondente tabella. Una parte di tale codice viene riorganizzato nella funzione `workBatches()` la quale si occupa, appunto, di bufferizzare tutte righe che vengono emesse dall'observable `works` nell'arco temporale di 500 millisecondi. La funzione prende come parametro in ingresso uno scheduler, è buona norma, infatti, parametrizzare gli scheduler in observable che dovranno essere testati.

Successivamente la funzione `workBatches()` viene testata mediante l'utilizzo di un testScheduler e al framework QUnit.

Inizialmente vengono caricate funzioni di supporto da `ReactiveTest` che permettono di registrare eventi 'onNext', 'onCopleted' e 'subscribe' in un tempo virtuale poi, viene creato TestScheduler.

Il metodo `createHotObservable` di TestScheduler permette di creare un hot observable fittizio che simula di notificare l'observer in determinati punti del tempo virtuale. In particolare, emette cinque notifiche nel primo secondo e completa al millisecondo 1100. Ogni volta viene emesso un oggetto con la proprietà `parentNode`. Il metodo `startScheduler` crea un observable che utilizza il testSheduler e prede come parametri in ingresso una funzione che restituisce l'Observable ottenuto dalla funzione `workBatches()` e, un oggetto che contiene informazioni circa il momento in cui deve avvenire la creazione, sottoscrizione ecc della sequenza.

`startScheduler` restituisce un oggetto con due proprietà: `scheduler`, `messages`. Da `messages` è possibile ottenere gli elementi emessi dalla sequenza. Infine, vengono fatte tre asserzioni per verificare se l'observable ha prodotto sequenze giuste.

```
function workBatches(scheduler){
    return works
        .pluck('parentNode')
        .bufferWithTime(500, null, scheduler || null)
        .filter(function(rows){
            return rows.length > 0
        });
}

var onNext = Rx.ReactiveTest.onNext;
var onCompleted = Rx.ReactiveTest.onCompleted;
var subscribe = Rx.ReactiveTest.subscribe;

var scheduler = new Rx.TestScheduler();

var works = scheduler.createHotObservable(
    onNext(100, {parentNode: 1}),
    onNext(300, {parentNode: 2}),
    onNext(550, {parentNode: 3}),
    onNext(750, {parentNode: 4}),
    onNext(1000, {parentNode: 5}),
    onCompleted(1100)
);

QUnit.test("Test work buffering", function(assert){
    var result = scheduler.startScheduler(function(){
        return workBatches(scheduler)
    } , {
        created:0,
        subscribed:0,
        disposed:1200
    });

    var messages = result.messages;

    assert.equal(
        messages[0].toString(),
        onNext(501, [1,2]).toString()
    );
```

```
    assert.equal(
        messages[1].toString(),
        onNext(1001, [3,4,5]).toString()
    );

    assert.equal(
        messages[2].toString(),
        onCompleted(1100).toString()
    );

});
```

Listato 3.10: test per la funzione workBatches()

Questo esempio (codice 3.10) mostra come sia facile testare codice asincrono mediante gli strumenti offerti da RxJS.

Test utente La web application è stata più volte testata e permette all'utente di eseguire tutte le funzionalità progettate mediante il paradigma di programmazione reattivo.

Mentre, per quanto riguarda l'implementazione delle funzionalità previste durante la fase di analisi e non esplicitamente trattate con la programmazione reattiva e quindi, implementate con il tradizionale approccio imperativo, l'unica funzionalità mancante è la gestione dell'archivio. L'utente ha la possibilità di visualizzare l'elenco dei casi, dei faldoni e degli utente ma, attraverso la web app, non è consentito anche il loro inserimento e/o modifica.

Capitolo 4

La programmazione reattiva a confronto con gli altri approcci per la programmazione asincrona

4.1 La gestione di codice asincrono

Gli approcci attualmente utilizzati per gestire la programmazione asincrona sono:

- task/futures based
- continuation passing style (CPS) e event-driven

task/futures based Mediante l'inizializzazione di una computazione asincrona(task), un oggetto rappresentante un risultato futuro o lo stato della computazione stessa, viene creato e immediatamente restituito. Tale oggetto consente di verificare lo stato del task (o del poll di task), bloccarsi quando si ha bisogno del suo risultato, cancellare quando possibile il task in corso, intercettare e gestire eventuali eccezioni o errori derivanti dalla sua esecuzione ecc.

Esempi di questo approccio sono: Java Task Executor framework, Microsoft Task-based Asynchronous Pattern, Grand-Central Dispatch in IOS, AsyncTask in Android.

continuation passing style (CPS) e event-driven In un programma continuation passing style quando viene attivata l'esecuzione di una computazione asincrona, viene specificata una funzione che deve essere richiamata come "continuo" nel momento in cui il calcolo asincrono è terminato o è avvenuto un errore; il controllo del programma viene passato esplicitamente nella forma di una continuazione. Con lo stile di programmazione ad eventi, invece, il controllo del programma è guida dagli eventi che avvengono nell'ambiente circostante, passerà dunque dall'esecuzione di un event handler ad un altro.

4.2 Codice asincrono in Javascript

In javascript per scrivere codice asincrono si fa uso di: callback, promises e eventi.

Callback Una callback è una funzione (A) passata come parametro ad un'altra funzione (B) che esegue un'operazione asincrona. Quando (B) è stata eseguita, essa richiama (A) passandole il risultato dell'operazione appena conclusa. Le callback sono spesso utilizzate per gestire flussi asincroni come l'I/O della rete, l'accesso a database, o l'input dell'utente.

```
function B(callback){
   //Do operation that takes some time
   callback('Done');
}

function A(message){
   console.log(message)
}

//Execute 'B' with 'A' as a callback
B(A);
```

Listato 4.1: Esempio callback

Utilizzare le callback può essere abbastanza intuitivo in situazioni relativamente semplici, ma può rapidamente trasformarsi in un incubo all'aumentare della complessità del codice.
Le callback, infatti, portano i seguenti svantaggi:

- scarsa leggibilità del codice che in breve può diventare affetto dalla cosiddetta Pyramid of Doom, l'espansione verso destra dovuta agli annidamenti indiscriminati delle callback ed alle relative indentazioni (callback hell).

- Difficoltà di composizione delle callback e di sincronizzazione del flusso di elaborazione, per ottenere i quali spesso occorre inventarsi artifici che rendono ancora più illeggibile il codice e talvolta poco efficiente, ad esempio, tenere traccia dello stato di ogni operazione in una variabile temporale e, successivamente combinare il loro risultato nel giusto ordine.

- Difficoltà di gestione degli errori e di debug, soprattutto in presenza di callback anonime. Che succede infatti quando si verifica un errore all'interno di una callback? Non è possibile gestire l'eccezione all'interno della funzione chiamante e questo può porre dei problemi non indifferenti.

- Difficoltà di gestire callback che vengono richiamate più volte, invocazioni multiple sono difficili da individuare e possono provocare errori e caos generale all'interno dell'applicazione.

A differenza delle chiamate a funzioni sincrone dalla cui esecuzione si attende un valore di ritorno o un'eccezione, nel caso di chiamate asincrone non si ha nessuna delle due cose. Di conseguenza viene meno la possibilità di composizione tra funzioni e la gestione delle eventuali eccezioni.

```
step1(function(result1){
   step2(function(result2){
      step3(function(result3){
      // and so on ....
      }
   }
}
```

Listato 4.2: Esempio di callback innestate o pyramid of doom

Eventi L'utilizzo della programmazione ad eventi, come le callback, ha dei problemi, vediamone alcuni:

- callback hell e asynchronous spaghetti. Il codice asincrono è frammentato all'interno dei vari gestori di eventi, ciò rende i programmi poco comprensibili e di difficile manutenibilità.

- Il valore di ritorno degli event handler viene sempre ignorato, questo può causare problemi quando determinati aspetti del sistema dipendono da questo.
- Gli eventi non sono oggetti di prima classe, ad esempio, una sequenza di eventi 'click' non può essere passata come parametro o manipolata come una collezione anche se lo è. Si è limitati a gestire ogni evento individualmente e, solo dopo che è avvenuto.
- E' facile perdere eventi se l'ascoltatore viene collegato alla sorgente troppo tardi.

Promise La soluzione ai problemi visti precedentemente potrebbe essere la programmazione per promesse (promise). Le promise, letteralmente promesse sono oggetti che rappresentano il risultato di una chiamata di funzione asincrona. Esse rappresentano una promessa che un risultato verrà fornito non appena disponibile (placeholder per il valore finale). Una promise può trovarsi in tre differenti stati: pending (non si è ancora ottenuta alcuna risposta dalla chiamata asincrona), resolved (la chiamata asincrona ha prodotto un risultato), rejected (non è possibile ottenere un risultato a causa di un errore). Si riporta un esempio in javascrip di promise.

```
//Creating a promise
var promise = new Promise(function(resolve, reject) {

   if (/* everything turned out fine */) {
      resolve("Stuff worked!");
   }
   else {
      reject(Error("It broke"));
   }
});

//Attaching the callback
promise.then(function(result) {
   console.log(result); // "Stuff worked!"
}, function(err) {
   console.log(err); // Error: "It broke"
});
```

Listato 4.3: Esempio di promise in javascript

Le promise sono una soluzione piuttosto elegante per risolvere il problema della callback innestate poiché forniscono un meccanismo per incapsularle e

per considerarle come oggetti di prima classe. Rendono, inoltre, il codice più leggibile, più simile al flusso di esecuzione sincrona e in alcune situazioni anche più efficiente. Tuttavia non tutto può essere modellato come una promessa:

- elaborazioni incrementali
- elaborazioni di eventi e sequenze

Le promesse non sono adatte, ad esempio, a gestire grandi flussi di dati proveniente da un server (SSE, WebSocket...), per gestire eventi ricorrenti dell'interfaccia grafica o per elaborazioni incrementali di grandi collezioni perché sarebbe necessario creare una promessa per ogni distinto evento/dato ricevuto invece di creare una promessa per l'intera sequenza.

I problemi derivanti dall'utilizzo in javascript dei tre elementi visti precedentemente (callback, eventi e promesse), coinvolgono anche tutti gli altri linguaggi di programmazione che ne fanno uso.

4.3 Codice asincrono con RxJS

In questa sezione si andranno a confrontare porzioni di codice scritte nel classico modo imperativo con lo stesso codice scritto mediante il relativo approccio reattivo, evidenziando eventuali punti di forza e/o debolezza.

Nell'esempio che segue si vogliono stampare le coordinate dei primi 10 click del mouse nella parte destra dello schermo. Nella prima versione (listato 4.4) di tale esempio è necessario introdurre una variabile globale, magari condivisa con altre funzioni, clicks, che mantiene il conteggio delle pressioni effettuate dall'utente. Sono inoltre necessari due blocchi condizionali e annullare la registrazione per non consumare troppa memoria. Tale codice è abbastanza complicato per un obiettivo così semplice, è difficile manipolarlo e, la sua comprensione non è immediata per uno sviluppatore che lo legge per la prima volta.

Nella seconda versione (listato 4.5) gli eventi di click sullo schermo vengono trattati come uno stream che è possibile interrogare e modificare. In questo caso il codice risulta facile da leggere, non compaiono variabili esterne che mantengono parte dello stato dell'esecuzione: il codice non ha dipendenze esterne e questo rende più difficile l'introduzione di bug. Anche se non avviene esplicitamente l'annullamento della registrazione degli event handlers, non si verificherà nessuno spreco di memoria poiché ciò avviene automaticamente dopo 10 click.

```
document.body.addEventListener('mousemove', function(e) {
   console.log(e.clientX, e.clientY);
});

var clicks = 0;
document.addEventListener('click', function registerClicks(e) {
   if (clicks < 10) {
      if (e.clientX > window.innerWidth / 2) {
         console.log(e.clientX, e.clientY);
         clicks += 1;
      }
   } else {
      document.removeEventListener('click', registerClicks);
   }
});
```

Listato 4.4: codice che si occupa di stampare a video i primi 10 click del mouse sulla destra dello schermo

```
Rx.Observable.fromEvent(document, 'click')
   .filter(function(c) { return c.clientX > window.innerWidth / 2; })
   .take(10)
   .subscribe(function(c) { console.log(c.clientX, c.clientY) })
```

Listato 4.5: codice scritto mediante RxJs che si occupa di stampare a video i primi 10 click del mouse sulla destra dello schermo

Se un'azione ha effetto anche fuori dal contesto in cui avviene, è considerata un effetto collaterale. Ad esempio, modificare il valore di una variabile locale che esiste solo all'interno di una funzione è sicuro, ma se la variabile è globale altre funzioni possono modificare il suo valore. Tale variabile potrebbe essere infatti modificata da altre funzioni e il suo valore diverso da quello che ci si può aspettare: in questo modo il codice risulta più complicato e incline agli errori. Tutto questo è necessario evitarlo soprattutto in programmi reattivi dove si hanno importanti frammenti di codice il cui valore cambia nel tempo.

Verranno ora riportati e confrontati frammenti di codice del caso di studio 'AvvocaTimer' visti in veste reattiva e non.

All'interno della home della web application 'Avvocatimer' si hanno tre bottoni rispettivamente per aggiornare la pagina, salvare le modifiche e stoppare il timer attivo su un lavoro. Nel codice riportato in (codice 4.6) ogni

bottone ha un proprio ascoltatore e tutte le volte che avviene l'evento 'click' viene richiamato un metodo di questo. Il codice per gestire il click dei tre bottoni viene, dunque, frammento all'interno di tre event handler diventando più difficile da comprendere e da mantenere. Con la programmazione reattiva (listato 4.8) i tre eventi vengono rappresentati con tre stream, invece di gestire un solo evento alla volta e possibile gestire una sequenza di eventi e, a differenza del modo imperativo, è possibile anche applicare operatori in catena a questi.

Lo stesso codice, dunque, in chiave reattiva risulta più semplice, chiaro ed elegante.

```
<!-- on refreshBtn click -->
function onRefreshBtnClick(){
    var xhttp = new XMLHttpRequest();
   //una volta ottenuta una risposta positiva dal server, viene
      effettuata una nuova richiesta per recuperare i lavori
      dell'utente
    xhttp.onreadystatechange = function() { ......... };
    xhttp.open("GET", Routing.generate('userWork'), true );
    xhttp.send();
}

<!-- on stopBtn click -->
function onStopBtnClick(){
    var params =
       "officeWorkId="+getOfficeWorkId()+"&note="+getNote()
       +"&workId="+getIdWork();
    var xhttp = new XMLHttpRequest();
   //una volta ottenuta una risposta positiva dal server, viene
      effettuata una nuova richiesta per recuperare i lavori
      dell'utente
    xhttp.onreadystatechange = function(){
       .......................... };
    xhttp.open("GET",
       Routing.generate('stopTimerSaveNoteAndWork')+"?"+params, true
       );
    xhttp.send();
}

<!-- on saveChangesBtn click -->
function onSaveChangesBtnClick(){
    var params =
       "officeWorkId="+getOfficeWorkId()+"&note="+getNote()+"&workId="+getIdWork();
```

```
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function(){};
    xhttp.open("GET",
        Routing.generate('saveNoteAndWork')+"?"+params, true );
    xhttp.send();
```

Listato 4.6: Event handlers per gestire gli eventi di pressione dei bottoni di aggiornamento, stop del timer e salvataggio di modifiche

Alla pressione dei tre bottoni occorre effettuare una richiesta al server per ottenere i lavori dell'utente. Come si può vedere nel frammento di codice (listato 4.7), quando, dopo la pressione del bottone salva, giunge una risposta positiva dal server indicando che le modifiche sono state apportate correttamente, occorre inoltrare una nuova richiesta per i lavori: tutto questo viene ripetuto per ogni evento 'click' sul bottone salva e, per ogni evento 'click' sui restanti due bottoni. La programmazione reattiva e i suo operatori permettono di fare tutto questo un sola volta evitando di ripeterlo nei vari event handler dei tre eventi in questione. Come mostrato in 4.8 i tre stream rappresentanti i tre eventi vengono uniti e, per ogni evento emesso da questi nell'arco temporale di 500 millisecondi, effettuata una richiesta al server. Ancora un volta il codice risulta più intuitivo e compatto, inoltre, se si volesse modellare il fatto che fra una richiesta e l'altra fatta al server per ottenere i lavori, devono passare almeno 500 millisecondi, sarebbe necessaria una variabile globale contente il tempo trascorso e dei controlli per verificare il suo valore. Grazie alla programmazione reattiva e ai suoi operatori è stato possibile, in questo caso, eliminare dipendenze esterne e ridurre l'utilizzo di strutture di controllo.

```
function onSaveChangesBtnClick(){
    var params =
        "officeWorkId="+getOfficeWorkId()+"&note="+getNote()
        +"&workId="+getIdWork();
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (xhttp.readyState == 4 && xhttp.status == 200){
            var http = new XMLHttpRequest();
            http.onreadystatechange = function() {
                if (http.readyState == 4 && http.status == 200){
                    document.getElementById("recentActivityTable").innerHTML
                        = http.response;
                }
            };
            http.open("GET", Routing.generate('userWork'), true );
            http.send(null);
```

```
        }
    };
    xhttp.open("GET", Routing.generate('saveNoteAndWork'), true );
    xhttp.send(null);

}
```

Listato 4.7: Callback innestate all'interno di un event handler

Riassumendo, nel frammento di codice (listato 4.8) gli eventi di pressione dei tre bottoni vengono modellati come stream, uniti attraverso l'operatore merge e, per ogni elemento da questi emesso, effettuata una richista ajax al server. Attraverso la programmazione reattiva, non sono state aggiunge nuove funzionalità ma, è stato possibile ottenere codice non frammentato e più compatto, semplice da comprendere e mantenere, privo di duplicazioni e dipendenze esterne e senza cicli condizionali.

```
var btnSaveChanges = $('#btn_saveChanges');
var btnStopTimer = $('#btn_stopTimer');
var btnRefresh = $('#btn_refresh');

var saveChangesClickStream = Rx.Observable.fromEvent(btnSaveChanges,
    'click')
   .flatMap(function(s){
      var s = "officeWorkId="+getOfficeWorkId()+"&note="+getNote()
          +"&workId="+getIdWork();
      return Rx.DOM.ajax({ url: pathcontrollerSaveChanges+"?"+s,
          responseType: 'json'})
   });

var stopTimerClickStream = Rx.Observable.fromEvent(btnStopTimer,
    'click')
   .flatMap(function(s){
      var s = "officeWorkId="+getOfficeWorkId()+"&note="+getNote()
          +"&workId="+getIdWork();
      return Rx.DOM.ajax({ url: pathControllerStopTimer+"?"+s,
          responseType: 'json'})
   });

var responseStream = Rx.Observable.fromEvent(btnRefresh, 'click')
   .merge(saveChangesClickStream)
   .merge(stopTimerClickStream)
   .debounce(500)
   .startWith('startup click')
```

```
.flatMap(function(){
  return Rx.DOM.ajax({ url: pathcontroller, responseType:
      'json'})
})
.retry(3);
```

Listato 4.8: Codice scritto con RxJs che permette la gestione di tre bottoni e di effettuare richieste al server.

Conclusioni

E' stato interessante e motivante studiare questo nuovo paradigma di programmazione e testarlo all'interno di un contesto applicativo reale. La riprogettazione di una parte significativa del caso di studio attraverso la programmazione reattiva mi ha permesso di comprendere appieno quali siano gli obiettivi principali e i benefici che si possono ottenere utilizzandola in sistemi altamente reattivi e composti per la maggiore da codice asincrono.

Ripensare le stesse funzionalità in termini di stream e di operatori applicabili a questi ha portato a focalizzarmi su cosa il sistema dovesse fare piuttosto che sul come realizzarle rendendo tutto il processo di progettazione e implementazione più semplice.
La seconda versione della web application, infatti, risulta molto più robusta rispetto alla prima, il suo codice più facile da comprendere e mantenere, compatto, elegante e senza alcuna dipendenza esterna.
E' bene sottolineare che con la programmazione reattiva non sono state aggiunte nuove funzionalità all'applicazione ma è stato migliorato e semplificato il modo con cui realizzarle.

Il sistema `AvvocaTimer`, in particolare, la web application, prevede quasi tutte le funzionalità richieste dallo studio legale che ha commissionato il progetto ad eccezione della gestione dell'archivio. L'utente può visualizzare casi, faldoni e clienti ma, non ha la possibilità di aggiungerne nuovi o eliminare quelli già esistenti.

Sviluppi Futuri

Attualmente il progetto è stato pensato per studi legali ma potrebbe essere ampliato ed esteso anche ad uffici o attività analoghe che abbiano le stesse necessità di tenere lo storico delle ore lavorate.
Si potrebbero, inoltre, prevedere dei moduli per il monitoraggio e la gestione delle presenze del personale e moduli che si occupino di analizzare i dati ed effettuare delle previsioni su di essi basandosi sui trends precedenti. Quest'ultimo sarebbe molto utile nel caso si voglia, per esempio, suggerire un avvocato ad un cliente basandosi sulle causa vinte nella materia del caso in questione o per altri scenari simili.
Questi possibili sviluppi rendono il sistema ancora più reattivo sarà perciò necessario gestire grandi porzioni di codice asincrono: in questi casi la programmazione reattiva può essere di aiuto per una più facile progettazione e implementazione di tali parti.

Ringraziamenti

Ringrazio i miei genitori poiché senza di loro non sarebbe stato possibile arrivare fino a questo punto.
Un grazie particolare va a Samantha e Filippo per aver fatto con me parte di questo progetto, lavorando senza sosta, giorno dopo giorno, sempre con il massimo impegno per provare a raggiungere l'obiettivo finale.
Ci tengo a ringraziare Francesco che mi ha offerto l'opportunità di lavorare ad un progetto concreto e anche tutti i suoi collaboratori per la pazienza e la disponibilità dimostrata durante la permanenza in azienda.
Ringrazio, infine, il mio relatore il prof. Alessandro Ricci per i preziosi consigli che mi ha fornito durante la stesura della tesi.

Bibliografia

[1] 2012, Bainomugisha, Lombide Carreton, Van Cutsem, Monstinckx, De Meuter.
A Survey on Reactive Programming

[2] 2012, Meijer.
Your mouse is a database

[3] https://github.com/Reactive-Extensions/RxJS.
Documentazione RxJS

[4] https://github.com/ReactiveX/RxPHP.
Documentazione RxPHP

[5] http://reactivex.io.
ReactiveX

[6] https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
The introduction to Reactive Programming you've been missing

[7] http://rxmarbles.com
Marble diagram

[8] 2015, Sergi Mansilla.
Reactive Programming with RxJS

[9] 2015, Alessandro Ricci.
Dispense del corso di Programmazione Avanzata e Paradigmi

[10] 2014, http://www.reactivemanifesto.org/.
Reactive Manifesto