ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA SEDE DI CESENA

SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

# PROGETTAZIONE E SVILUPPO DI APPLICAZIONI DISTRIBUITE SU WEB: DA JAVASCRIPT A TYPESCRIPT

Tesi in:
Programmazione Concorrente e Distribuita LM

*Presentata da*:
ENRICO GRAMELLINI

*Relatore*:
Prof. ALESSANDRO RICCI

*Correlatore*:
Dott. ANDREA SANTI

ANNO ACCADEMICO 2011/2012
SESSIONE III

# PAROLE CHIAVE

**Web Application**

**Typescript**

**Javascript**

Dedicata alla mia famiglia, ai miei amici, ai miei colleghi ed a tutti quelli che mi vogliono bene, ma soprattutto a Simone e Marica.

# Indice

# Introduzione

## Scenario

Il Web nel corso della sua esistenza ha subito un mutamento dovuto in parte dalle richieste del mercato, ma soprattutto dall'evoluzione e la nascita costante delle numerose tecnologie coinvolte in esso. Si è passati da un'iniziale semplice diffusione di contenuti statici, ad una successiva collezione di siti web, dapprima con limitate presenze di dinamicità e interattività (a causa dei limiti tecnologici), ma successivamente poi evoluti alle attuali applicazioni web moderne che hanno colmato il gap con le applicazioni desktop, sia a livello tecnologico, che a livello di diffusione effettiva sul mercato.

Tali applicazioni web moderne possono presentare un grado di complessità paragonabile in tutto e per tutto ai sistemi software desktop tradizionali; le tecnologie web hanno subito nel tempo un evoluzione legata ai cambiamenti del web stesso e tra le tecnologie più diffuse troviamo JavaScript, un linguaggio di scripting nato per dare dinamicità ai siti web che si ritrova tutt'ora ad essere utilizzato come linguaggio di programmazione di applicazioni altamente strutturate.

Nel corso degli anni la comunità di sviluppo che ruota intorno a JavaScript ha prodotto numerose librerie al supporto del linguaggio dotando così gli sviluppatori di un linguaggio completo in grado di far realizzare applicazioni web avanzate. Le recenti evoluzioni dei motori javascript presenti nei browser hanno inoltre incrementato le prestazioni del linguaggio consacrandone la sua leadership nei confronti dei linguaggi concorrenti.

Negli ultimi anni a causa della crescita della complessità delle applicazioni web, javascript è stato messo molto in discussione in quanto come linguaggio non offre le classiche astrazioni consolidate nel tempo per la programmazione altamente strutturata; per questo motivo sono nati linguaggi orientati

alla programmazione ad oggetti per il web che si pongono come obiettivo la risoluzione di questo problema: tra questi si trovano linguaggi che hanno l'ambizione di soppiantare JavaScript come ad esempio Dart creato da Google, oppure altri che invece sfruttano JavaScript come linguaggio base al quale aggiungono le caratteristiche mancanti e, mediante il processo di compilazione, producono codice JavaScript puro compatibile con i motori JavaScript presenti nei browser.

JavaScript storicamente fu introdotto come linguaggio sia per la programmazione client-side, che per la controparte server-side, ma per vari motivi (la forte concorrenza, basse performance, etc.) ebbe successo solo come linguaggio per la programmazione client; le recenti evoluzioni del linguaggio lo hanno però riportato in auge anche per la programmazione server-side, soprattutto per i miglioramenti delle performance, ma anche per la sua naturale predisposizione per la programmazione event-driven, paradigma alternativo al multi-threading per la programmazione concorrente.

Un'applicazione web di elevata complessità al giorno d'oggi può quindi essere interamente sviluppata utilizzando il linguaggio JavaScript, acquisendone sia i suoi vantaggi che gli svantaggi; le nuove tecnologie introdotte ambiscono quindi a diventare la soluzione per i problemi presenti in JavaScript e di conseguenza si propongono come potenziali nuovi linguaggi completi per la programmazione web del futuro, anticipando anche le prossime evoluzioni delle tecnologie già esistenti preannunciate dagli enti standard della programmazione web, il W3C ed ECMAScript.

## Obiettivi

In questa tesi saranno affrontate le tematiche appena introdotte confrontando tra loro le tecnologie in gioco con lo scopo di ottenere un'ampia panoramica delle soluzioni che uno sviluppatore web dovrà prendere in considerazione per realizzare un sistema di importanti dimensioni; in particolare sarà approfondito il linguaggio TypeScript proposto da Microsoft, il quale è nato in successione a Dart apparentemente con lo stesso scopo, ma grazie alla compatibilità con JavaScript e soprattutto con il vasto mondo di librerie legate ad esso nate in questi ultimi anni, si presenta nel mercato come tecnologia facile da apprendere per tutti gli sviluppatori che già da tempo hanno sviluppato abilità nella programmazione JavaScript.

Nel primo capitolo sarà illustrata l'evoluzione delle applicazioni web, dalla nascita del web allo stato attuale dell'arte, dando maggiore enfasi al recente passato legato alla storia del web dinamico e di conseguenza del linguaggio JavaScript.

Nel secondo e nel terzo capitolo l'enfasi si sposterà sull'aspetto architetturale e tecnologico, cioè come sono evoluti i modelli architetturali di riferimento e le principali tecnologie utilizzate per lo sviluppo in fronte ai cambiamenti delle esigenze presentate dalle applicazioni web; saranno analizzati i nuovi requisiti che presentano le applicazioni moderne e i cambiamenti nel processo di ingegnerizzazione di tali sistemi che hanno ormai raggiunto elevati livelli di complessità.

Nel quarto capitolo saranno introdotte alcune nuove tecnologie nate per lo sviluppo di applicazioni web moderne, in particolare Node.js, Dart e TypeScript: si tratta relativamente di una piattaforma per lo sviluppo di applicazioni web interamente scritte in JavaScript e di due linguaggi orientati alla programmazione ad oggetti per il web.

Nel quinto capitolo sarà approfondito maggiormente in dettaglio il linguaggio TypeScript intorno al quale ruota l'enfasi della tesi e sarà confrontato con i linguaggi Dart e JavaScript.

Nel sesto ed ultimo capitolo sarà illustrato un caso di studio applicativo riguardante un sistema realizzato in collaborazione con la software house TwinLogix s.r.l. utilizzando alcune di queste nuove tecnologie analizzate. L'applicativo consisterà in un Social Network dedicato agli utenti di circoli tennis affiliati con lo scopo di tracciare partite di Tennis e Beach Tennis organizzate dagli utenti attraverso l'applicazione web; tale sistema classificherà annualmente gli utenti in base ai risultati da loro conseguiti e a fine stagione sarà così possibile organizzare tornei finali ai quali potranno partecipare i migliori giocatori di ciascuno sport per categoria e sesso appartenente. L'applicazione web sarà realizzata in particolare utilizzando la piattaforma Node.js che permetterà lo sviluppo dell'intera applicazione utilizzando il linguaggio JavaScript sia lato client, che lato server. Sarà inoltre utilizzato il linguaggio TypeScript per strutturare più ad alto livello il codice sorgente, il quale poi attraverso il processo di compilazione sarà trasformato in JavaScript puro per poter essere compatibile con le piattaforme che dovranno eseguire il sistema.

# Capitolo 1

# Evoluzione delle Applicazioni Web

La maggior parte dei sistemi software moderni sfruttano il web per avere caratteristiche di networking (mobilità, connettività, distribuzione di dati e/o di computazione). Nell'era del Cloud Computing è nato il concetto di Software as a Service (SaaS), un nuovo modo di concepire il software dal punto di vista della fruizione del prodotto finale che, insieme alle altre innovazioni tecnologiche, hanno contribuito notevolmente sia ad aumentare il processo di informatizzazione delle aziende, che a mutare il mercato del software, fin'ora dominato dai sistemi desktop ormai superati dai sistemi basati sul web.

L'evoluzione del web è stato quindi un fattore fondamentale anche per l'evoluzione stessa dei sistemi software; in questo capitolo analizzando l'evoluzione del web, saranno catturati gli aspetti riguardanti l'influenza che essa ha avuto nei confronti dei sistemi software e viceversa, cioè come è evoluto il web in fronte alle evoluzioni dei sistemi software e del loro mercato.

## 1.1 Web 1.0

Il Web (ellissi di World Wide Web) è il principale servizio di Internet che permette di navigare ed usufruire di un insieme vastissimo di contenuti (multimediali e non) e di ulteriori servizi accessibili a tutti o ad una parte selezionata degli utenti di Internet. [1]

Nacque nel 1989 come progetto per migliorare la condivisione di informazioni tra scienziati all'interno del laboratorio del CERN di Ginevra: la

soluzione proposta dallo scienziato inglese Tim Berners-Lee consisteva nella possibilità di condividere documentazione scientifica in formato elettronico in modo indipendente dalla piattaforma informatica utilizzata. Tale software venne affiancato dalla definizione di standard e protocolli per scambiare i documenti all'interno di reti di calcolatori: il linguaggio HTML e il protocollo di rete HTTP.

La data di nascita del World Wide Web viene comunemente indicata nel 6 agosto 1991, giorno in cui venne pubblicato il primo sito web, mentre il 30 aprile 1993 il CERN decise di mettere il Web a disposizione del pubblico rinunciando ad ogni diritto d'autore. La semplicità della tecnologia decretò un immediato successo: in pochi anni il Web divenne la modalità più diffusa al mondo per inviare e ricevere dati su Internet, facendo nascere quella che oggi è nota come era del web.

Questi standard e protocolli supportavano inizialmente la sola gestione di pagine HTML statiche, ma fin da subito furono introdotte le prime tecnologie in grado di generare pagine HTML in modo dinamico: le prime furono le CGI (Common Gateway Interface), attraverso le quali era possibile richiedere ad un web server l'esecuzione di un'applicazione esterna precedentemente compilata e al termine della sua esecuzione restituiva al client il risultato ottenuto, cioè la pagina HTML da visualizzare. Con questa tecnologia il web dinamico era comunque molto limitato e in pochi anni nacquero ulteriori tecnologie che permisero di superare molti di questi limiti; le evoluzioni presero due strade: da un lato aumentarono le funzionalità dei browser attraverso un'evoluzione del linguaggio HTML e la possibilità d'interpretazione di linguaggi di scripting (come il JavaScript); dall'altro migliorò la qualità di elaborazione dei server attraverso una nuova generazione di linguaggi integrati con il Web Server (come JSP, PHP, ASP, ecc.), trasformando questi ultimi in quelli che sono oggi più propriamente noti come Application Server.

Con questa nuova versione del Web dinamico crebbe la diffusione di applicazioni orientate al business ed ai consumatori attraverso internet e quindi non più solo sulla computazione locale. L'architettura più diffusa per queste applicazioni prevedeva tipicamente 3 strati: presentazione, contenuto e dati, con carico computazionale prevalentemente lato server, strato installato su macchine molto performanti. Le applicazioni del Web 1.0 ancora però mostravano un gap tecnologico da colmare nei confronti delle applicazioni desktop, sia in termini di performance (siti piuttosto lenti, frequenti refresh necessari per la gestione della dinamicità, gestione scarna delle sessioni, ecc.),

ma soprattutto in termini di interattività. Nascono in questo periodo storico i motori di ricerca, ma i primi non ebbero successo soprattutto a causa della loro inefficienza (basati puramente sulle dimensioni degli indici e non sulla rilevanza) che addirittura non permetteva neanche di trovare loro stessi.

Il Web 1.0 si può riassumere come una collezione di siti web, alcuni ancora con contenuti puramente statici, ma molti già dotati di un alta percentuale di dinamicità. Nel periodo storico 97'-2000 circa, esplosero attorno al web interessi economici tali da far raggiungere un picco nell'economia informatica mai più raggiunto da quel momento fino ad oggi. Raggiunto il picco però avvenne il "collasso del dot com" nel quale solo alcune aziende sopravvissero (Amazon, Google, ecc.): sono state probabilmente quelle con le basi più solide, con un modello di business che ha saputo evolversi e tenere il passo, dato che fin dal principio le potenzialità del web dinamico non furono comprese e la maggior parte delle applicazioni create seguivano i vecchi modelli di business applications.

## 1.2 Web 2.0

> "Il Web 2.0 è un termine utilizzato per indicare uno stato di evoluzione del World Wide Web, rispetto a una condizione precedente. Si indica come Web 2.0 l'insieme di tutte quelle applicazioni online che permettono uno spiccato livello di interazione tra il sito web e l'utente (blog, forum, chat,wiki, flickr, youtube, facebook, myspace, twitter, google+, linkedin, wordpress, foursquare, ecc.) ottenute tipicamente attraverso opportune tecniche di programmazione Web afferenti al paradigma del Web dinamico in contrapposizione al cosiddetto Web statico o Web 1.0." [1]

La definizione presente su Wikipedia sottolinea come differenza principale dal Web 1.0 alla versione successiva l'aumento del livello di interazione presente all'interno delle applicazioni web. I termini coniati per classificare il Web sono nati contemporaneamente alla nascita del Web 2.0, appunto per sottolineare un cambiamento radicale. Il Web 2.0 più che qualcosa di nuovo è una piena realizzazione del vero potenziale della piattaforma web che finora non era stata raggiunta.

Tim O'Reilly ("creatore" del termine Web 2.0) afferma nel 2005: "Il Web 2.0 è la rete intesa come piattaforma che sfrutta tutti i dispositivi collegati; le

applicazioni Web 2.0 sono quelle che utilizzano la maggior parte dei vantaggi intrinseci di tale piattaforma: la distribuzione del software diventa un servizio in continuo aggiornamento che migliora all'aumentare delle persone che lo usano, consumando dati e mischiandoli da più fonti, tra le quali gli utenti stessi, fornendo propri dati e servizi in una forma che ne permette l'utilizzo da parte di altri, la creazione di effetti di rete attraverso un "architettura della partecipazione", andando oltre la metafora del Web 1.0 per offrire esperienze utente sempre più ricche." [2]

### 1.2.1 Principi del Web 2.0

In questo passaggio di evoluzione del Web si possono individuare i principali principi che caratterizzano il Web 2.0:

- *Web inteso come piattaforma*: nell'era del software desktop Microsoft riuscì nel tempo a diventare la piattaforma leader nel mercato e di conseguenza essendo una piattaforma proprietaria, il singolo approccio monolitico da soluzione divenne un problema. La diffusione dei sistemi communication oriented che richiedono per loro natura interoperabilità, ha sancito la consacrazione del Web come principale piattaforma software antagonista di Microsoft. Si aprirono le strade per il successo di società che proponevano software sfruttando la piattaforma aperta del Web (Google, Ebay, Napster, ecc.), piuttosto che nuove piattaforme con lo scopo di soppiantare quelle già esistenti (Netscape, ecc.).

- *Sfruttamento dell'intelligenza collettiva*: il contributo degli utilizzatori delle applicazioni web è fondamentale nel Web 2.0 ed è diventata la chiave per il dominio del mercato nella seconda era del Web. Il ruolo della società è quello di mettere a disposizione un contesto in cui l'attività degli utenti possa aver luogo. Le società che vantano i più grandi successi in internet utilizzano il passaparola dei loro utilizzatori (marketing virale) come maggiore fonte di pubblicità.

- Le raccolte dati (*database*) sono un componente fondamentale per il business delle applicazioni Web 2.0, infatti il valore di tali applicazioni è diventato proporzionale alla scala e al dinamismo dei dati che esso aiuta a gestire. È possibile sfruttare sistemi per l'aggregazione dei dati degli utenti e per la costruzione di valore come effetto laterale dell'utilizzo

ordinario dell'applicazione. La corsa è per la proprietà di certe classi di dati centrali: molte società hanno fatto dei loro dati prodotti un vero e proprio punto forte nella concorrenza sul mercato, alcune producendo dati da zero, altre partendo da basi dati esistenti estendendole poi con ricchi dati aggiuntivi.

- *Fine del classico ciclo di vita del software*: i nuovi sistemi sono soggetti a evoluzioni costanti (beta perpetua), quindi con costanti e veloci rilasci di aggiornamenti e valutazioni sul successo delle novità introdotte in base all'apprezzamento degli utenti. Anche il modello di business aziendale è evoluto nel tempo, andando verso un sistema di aggiornamento quotidiano dell'ambiente informatico aziendale rispetto alle vecchie tempistiche (annuali).

- *Modelli di business leggeri*: nel Web 2.0 si è diffusa la creazione di software sotto forma di servizi (SaaS) ed allo stato attuale hanno avuto maggiore riscontro servizi web leggeri a discapito di stack di servizi complessi. Avendo a disposizione sempre più servizi web leggeri, è possibile raggiungere i propri scopi componendo tali servizi tra loro, piuttosto che creare tutto dal principio.

- Applicazioni basate sul concetto di *multi-device*: con l'aumento dei dispositivi dotati di intelligenza che non si occupano più solo di consumare informazioni, ma anche di produrle, è possibile creare sistemi software sfruttando la piattaforma web che si basano principalmente sulla gestione dei dati e sulla loro fruizione da diversi dispositivi: il primo caso di successo per un'applicazione di questo genere fu iTunes, che propose per prima una soluzione software in grado di sfruttare il proprio PC e un lettore multimediale (iPod) per gestire materiale acquistato attraverso servizi web su un massiccio back-end (Apple Store) in modo trasparente: in questo caso il PC e il lettore veniva sfruttato come cache locale e come stazione di controllo dei dati utilizzabili. La crescita del Web nel tempo è rappresentabile graficamente con una curva asintotica per sottolineare quanto all'inizio il web evolvesse lentamente e quanto l'aumento della connettività invece ha fatto aumentare più velocemente l'evoluzione negli ultimi anni.

- *Rich Internet Applications* (RIA): il termine RIA fu coniato dai creatori della tecnologia Flash. L'obiettivo di dare all'utente la possibilità di

avere una ricca user experience è alla base dell'evoluzione del web. L'evoluzione dei dispositivi comunemente usati e diffusi dotati di browser e l'evoluzione dei browser stessi ha portato il successo della computazione lato client con protagonista assoluto JavaScript, grazie anche ai miglioramenti nelle performance (avvenuti circa ogni 9 mesi negli ultimi 5 anni) per quanto riguarda la sua esecuzione nei motori presenti nei browser. JavaScript è la tecnologia che ha permesso il salto di qualità maggiore da parte delle applicazioni web, ma già dal principio tecnologie come le Applet furono inventate per ottenere questo scopo. Con questi strumenti le web application hanno potuto quindi avvicinarsi sempre di più ai risultati già ottenuti negli anni passati con le desktop applications; i primi casi di successo provengono da Google (Gmail, Google Maps, ...) il quale, utilizzando diverse tecnologie tra quelle più all'avanguardia, ha coniato con il termine AJAX, così definito su Wikipedia: "è un insieme di tecnologie (JavaScript, XML, HTML+CSS, DOM, XMLHttpRequest, ecc.) che, utilizzate insieme, permettono lo sviluppo di applicazioni web con ricche presenze di interattività'.' Queste tecnologie erano esistenti già da tempo, ma a causa della guerra tra i browser per accaparrarsi gli standard, solo nel momento in cui Microsoft ebbe la meglio su Netscape questa innovazione poté avere luogo. Attualmente la moderna concorrenza tra browser non sta causando però gli stessi effetti negativi. L'evoluzione di una nuova piattaforma è anche causa di un cambio di leadership nelle varie tipologie di servizi offerti, ad esempio il caso Gmail per app di gestione email. La ricchezza delle applicazioni del web 2.0 è quindi rappresentata in gran parte dall'interattività con e tra gli utenti e di conseguenza le applicazioni potranno anche apprendere dagli utenti attraverso un'architettura partecipativa, per costruire un vantaggio competitivo non solo nell'interfaccia software, ma anche nella ricchezza dei dati condivisi.

### 1.2.2 Rich Internet Applications

Nel precedente paragrafo, sono state introdotte le RIA, cioè le applicazioni tipiche del Web 2.0 che possiedono caratteristiche molto differenti dalle applicazioni del Web 1.0. In questo paragrafo saranno sottolineate le diffe-

renze esistenti rispetto alle applicazioni web tradizionali, valutandone aspetti architetturali e tecnologici.

L'usabilità (o esperienza d'uso) è in generale un aspetto fondamentale per lo sviluppo software ma per le applicazioni internet lo è diventato ancora di più data la vastità di tipologie di utilizzatori esistenti al mondo, ognuno con una sua cultura e conoscenza tecnica. Un'applicazione web che offre un'esperienza d'uso avanzata rimane ben impressa nella mente dell'utilizzatore e quindi quest'ultimo è maggiormente portato a ritornare ad utilizzarla invece che provare ad utilizzare un'altra applicazione di un competitor che invece offre lo stesso servizio ma non incentrato sull'esperienza d'uso. Le applicazioni internet tradizionali erano basate sulla classica architettura client-server semplificata il più possibile, dove nel back-end (server) risiede la maggior parte della computazione e nel front-end (client) risiede solamente la parte di rendering delle informazioni da visualizzare. A livello di esperienza d'uso si presentavano pagine che in fronte alla necessità di un aggiornamento dei dati all'interno, richiedevano il refresh dell'intera applicazione. Le RIA fanno di questi due fattori i loro migliori punti di loro forza, presentando elevata interattività con l'utilizzatore sfruttando maggiormente la computazione client side.

In base alle tecnologie utilizzate per lo sviluppo di questo tipo di applicazioni, le RIA si possono distinguere in 3 categorie: [3]

- *Plug-in based*: sviluppate in una piattaforma (Flash, Java, ecc.) che permette lo sviluppo di applicazioni avanzate ed eseguite/inserite direttamente nell'ambiente internet.

- *Script based*: basate su architettura client-server e su tecnologia AJAX, dove lato client mediante linguaggio di scripting (JavaScript) vengono inoltrate richieste in modo asincrono verso il server, il quale (sviluppato in una qualsiasi tecnologia server side) elabora le richieste: in seguito alla risposta inoltrata verso il client, sempre con il linguaggio di scripting sono renderizzati i dati "on the fly" nella pagina senza dover effettuare un refresh. Questa tipologia però presenta una maggiore difficoltà di sviluppo, in quanto le tecnologie utilizzate comportano una serie di problematiche che obbligano lo sviluppatore ad esserne innanzitutto a conoscenza e di conseguenza a prendere le dovute precauzioni cercando di sviluppare così un'applicazione completamente accessibile: ad esempio alcuni browser permettono di disabilitare il motore javascript e di

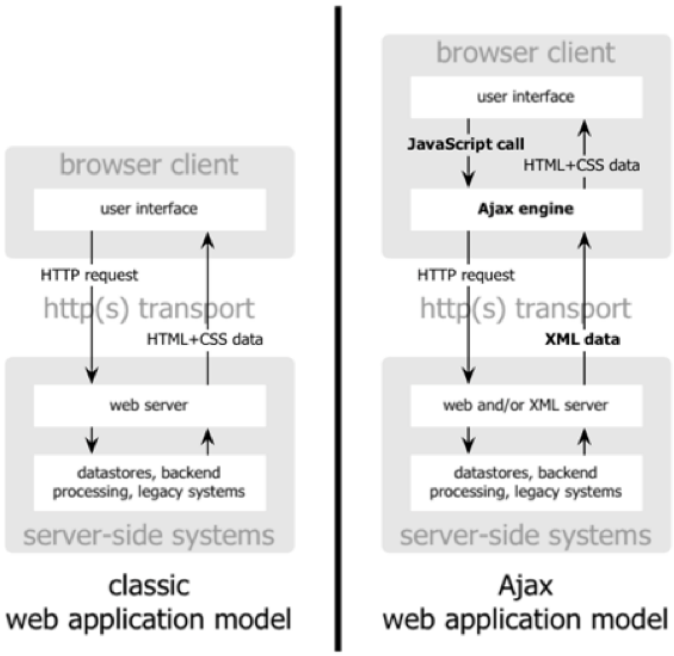


Figura 1.1: Modelli applicativi a confronto: Traditionali vs. Rich (Ajax) Internet.

conseguenza un'applicazione sviluppata con tale linguaggio di scripting dovrà fornirne una versione alternativa che probabilmente non offre la stessa esperienza d'uso, ma che almeno offra utilizzabilità.

- *Browser based*: sviluppate utilizzando un linguaggio XML based (XUL, ecc.) che permette di definire in maniera dichiarativa quali componenti di interfaccia devono essere presenti e quali sono le loro interazioni. Questi linguaggi sono platform independent per quanto riguarda la portabilità.

Tutte e tre le tipologie hanno comunque in comune la possibilità di ottenere dati in modo asincrono e di aggiornare la pagina "on the fly".

In figura 1.1 sono rappresentate le differenze tra i due modelli di web applications ed è possibile apprezzare come la differenza maggiore sia proprio quel blocco concettuale posto lato client per interfacciare quest'ultimo con il lato server.

Nelle RIA il processo di caricamento dell'interfaccia iniziale è identico a quello presente nelle web app tradizionali (tramite chiamate XMLHttpRequest), con l'aggiunta della fase di ajaxify della pagina, cioè il setting delle funzionalità javascript aggiuntive (event handling con manipolazione del DOM) che permettono alla pagina di avere un "anima dinamica" anche lato client. Tra gli obiettivi principali che le RIA hanno cercato di risolvere si collocano sicuramente le problematiche legate alle operazioni svolte dagli utenti attraverso la user interface: mentre un'operazione nelle web app tradizionali comportava un refresh dell'applicazione con tempistiche legate alla durata stessa dell'operazione, nelle RIA è possibile rimanere all'interno della stessa pagina aggiungendo effetti visuali che mostrano all'utente lo stato di avanzamento dell'operazione.

Un aspetto negativo legato alle RIA è che non ci sono architetture standard per il loro sviluppo e di conseguenza vengono seguiti diversi principi in base alla tecnologia che si utilizza; nonostante ciò alcune si sono più diffuse rispetto ad altre come ad esempio l'architettura REST, basata sui concetti di dati e servizi, è considerata potente e di facile utilizzo.

### 1.2.3 Cloud Computing

Il cloud computing può essere definito come un ambiente informatico in cui le esigenze di elaborazione possono essere esternate al sistema e quando sorge il bisogno di utilizzare potenza di calcolo o risorse come database, ci si può accedere via Internet. Il cloud computing è una tendenza recente che sposta computazione e dati dai PC desktop e portatili a grandi data center. Il principale vantaggio del cloud computing è la possibilità di ottimizzare i costi limitandoli all'effettivo utilizzo delle risorse, sia queste siano hardware, software o piattaforme.

Nell'era del Cloud Computing tutto ruota intorno ai "servizi":

- *Software as a Service* (SaaS): il software grazie alla piattaforma web non è più limitato a risiedere all'interno di PC desktop, con abbattimento conseguente di costi di distribuzione, licenze di utilizzo, installazione, ecc. Con l'avvento dei Web Service, nascono anche nuove modalità di fruizione del software, in quanto l'utilizzo di un servizio è ottimizzabile in base all'effettivo consumo. Si aprono le porte anche a nuove modalità di creazione di software: partendo da servizi già esistenti e

componendoli tra loro è possibile realizzare nuovi sistemi con il minimo sforzo, senza dover sviluppare completamente tutto l'insieme come accadeva invece prima dell'era dei web service. Inoltre in un sistema software l'esposizione di servizi per interfacciarsi ad esso, diventa anche un fattore che può velocizzare il processo di acquisizione di notorietà.

- *Data as a Service* (DaaS): l'importanza dei dati nell'era moderna del Web ha contribuito nella diffusione di servizi orientati ai dati, cioè vere e proprie banche dati messe a disposizione degli utilizzatori sotto forma di servizi.

- *Platform as a Service* (PaaS): i servizi in questo caso sono intere piattaforme di elaborazione. Un PaaS è una soluzione integrata sul Web che offre servizi che permettono di sviluppare, testare, implementare e gestire le applicazioni aziendali senza i costi e la complessità associati all'acquisto, la configurazione, l'ottimizzazione e la gestione dell'hardware e del software di base.

- *Infrastructure as a Service* (IaaS): sono servizi che offrono la possibilità di utilizzare risorse hardware su richiesta al momento in cui una piattaforma ne ha bisogno, risparmiandone i costi limitandone l'utilizzo effettivo. Si abbattono così le difficoltà nello stimare le risorse necessarie ad un sistema ed eventuali errori cause di perdite di denaro e di degrado delle prestazioni del sistema.

Legati al cloud computing ci sono però anche diversi fattori negativi riguardo a rischi nei quali un utilizzatore dei servizi può incorrere:

- *Sicurezza e Privacy*: la memorizzazione di dati su data center non proprietari espone gli utenti ad un possibile utilizzo scorretto dei propri dati da parte delle aziende che li possiedono; anche i collegamenti wireless altamente diffusi come mezzo per interfacciarsi ad internet, espongono a loro volta l'utilizzatore di servizi ad atti di pirateria informatica.

- *Economia e Politica*: le diverse legislazioni tra nazioni, soprattutto quando c'è molto divario di ricchezza, rendono difficoltoso e rischioso l'utilizzo di servizi appartenenti ad un paese diverso dal proprio, in quanto le legislazioni sono soggette a cambiamenti e non viene data nessuna garanzia all'utilizzatore sul libero accesso futuro.

- *Continuità del servizio*: l'utilizzo di servizi esterni non garantisce la funzionalità di essi all'infinito e, nel caso in cui un servizio non dovesse più essere fruibile, si è costretti a modificare il sistema prodotto.
- *Migrazione dei dati*: nel caso in cui si voglia cambiare servizio di gestione dei dati, ci si trova di fronte al problema della mancanza di standard che rende difficoltoso il processo di migrazione da un gestore ad un altro.

# Capitolo 2

# Evoluzione delle Architetture

Prima della nascita del web, il mercato del software era dominato da applicazioni per piattaforme desktop, ma con l'avvento del web e con le evoluzioni delle tecnologie e dei dispositivi comunemente utilizzati, nel corso degli ultimi anni si sono aperte le strade per sistemi software che sfruttano diverse piattaforme (web, mobile, ecc.): le società che decidono di investire nella IT, in base al sistema che devono realizzare hanno l'obiettivo di coprire una particolare fetta di mercato e di conseguenza un sistema software moderno tipicamente è multi-piattaforma sfruttando la rete internet come medium di comunicazione.

In questo capitolo saranno analizzate le evoluzioni avvenute nel mondo dello sviluppo software in seguito all'avvento del web, prendendo come riferimento temporale lo stato dell'arte pre-era del Web 2.0, confrontandolo con lo stato dell'arte attuale, evidenziando le evoluzioni architetturali del software.

## 2.1 Deskop Applications e Web Applications

La scelta di una piattaforma per la quale realizzare un software dipende principalmente da due fattori: il business di riferimento e le implicazioni di carattere tecnologico legate alla scelta stessa della piattaforma. Di seguito saranno elencate le principali differenze che si presentano di fronte alla scelta tra una piattaforma Desktop e la piattaforma Web:

- *Vincoli di utilizzo*: mentre una Desktop Application è vincolata ad essere utilizzata all'interno del dispositivo nel quale è installata, una

Web Application è utilizzabile in qualsiasi dispositivo dotato di browser collegato alla rete sulla quale è distribuita (Internet, Intranet, ecc.)

- *Mantenimento del software*: un'applicazione desktop necessita l'istallazione in ogni dispositivo sul quale deve essere eseguita, mentre un'applicazione web necessita solamente la prima installazione sul web server che la ospita; per quanto riguarda gli upgrade, anche in questo caso vale lo stesso concetto, ma le conseguenze non sono solo positive, in quanto un eventuale aggiornamento non desiderato dall'utilizzatore o addirittura contenente un malfunzionamento, nelle applicazioni desktop può essere ignorato, mentre nelle applicazioni web la decisione spetta al creatore del software e non all'utilizzatore finale.

- *Sicurezza*: in ambiente desktop si ha controllo totale su un applicazione e di conseguenza ci si può proteggere dalle vulnerabilità; in ambiente web le applicazioni sono esposte invece a maggiori rischi e l'utilizzatore deve fidarsi di ciò che utilizza.

- *Connettività*: le applicazioni web sono strettamente basate sulla connettività, in quanto l'assenza di connessione ad internet implicherebbe l'impossibilità di utilizzare l'applicazione da parte dell'utente; le applicazioni desktop invece per loro natura sono indipendenti dalla rete internet e in mancanza di connettività soffrirebbero solamente l'impossibilità di utilizzare particolari funzioni; la connettività influenza anche le prestazioni di tali funzionalità e nel caso di un'applicazione web una bassa velocità di connessione equivale ad un degrado delle performance delle operazioni eseguibili.

- *Costi*: la realizzazione ed il mantenimento di applicazioni web sono generalmente più costose e impegnative rispetto ad un'applicazione desktop, in quanto il tipico ciclo di sviluppo in un caso consiste in continui aggiornamenti, mentre nel secondo caso spesso dopo il primo rilascio il software viene aggiornato molto meno frequentemente. Dal punto di vista dell'utilizzatore invece, riguardo le applicazioni che prevedono un pagamento per i servizi offerti, nel caso delle applicazioni desktop la metodologia di fruizione più diffusa prevede un primo acquisto dopo il quale tipicamente non sono più richiesti costi aggiuntivi, mentre in ambiente web viene utilizzata maggiormente la modalità a sottoscrizioni, dove l'utente periodicamente effettua i pagamenti richiesti.

- *Performance*: mentre per le applicazioni desktop le performance di esecuzione sono completamente dipendenti dall'hardware presente sulla macchina operativa, nella controparte web le performance dipendono in parte dall'hardware che permette l'esecuzione del browser, ma soprattutto dai ritardi dovuti dal trasferimento dati attraverso internet: tali performance sono influenzate sia dalla velocità della connessione dell'utilizzatore, ma anche dal numero di utilizzatori che accedono contemporaneamente all'applicazione, in quanto il web server deve elaborare tutte le richieste che gli si presentano.

## 2.2 Modello client/server di riferimento

Rispetto alle applicazioni web tradizionali, le applicazione web moderne si distinguono per l'evoluta interattività e le maggiori funzionalità offerte all'utente. Uno dei modelli più diffusi nelle applicazioni Web 2.0 è quello orientato ai servizi (SOA - Service Oriented Architecture), il quale permette alle applicazioni web di esporre le proprie funzionalità attraverso servizi web: questo modello facilità l'operabilità con altri modelli applicativi differenti. In figura 2.1 è mostrato il modello di riferimento astratto per le applicazioni web moderne suddiviso in cinque strati fondamentali, ognuno dei quali è opportunamente esteso in sotto-strati in base alla tipologia di applicazione da realizzare.

Il Web 2.0 ha rivoluzionato sia lo strato server, in quanto alle applicazioni web si interfacciano al giorno d'oggi diverse varietà di dispositivi, ma soprattutto lo strato client, il quale ora mostra nuovi aspetti: uno strato applicativo, una fase di esecuzione (runtime) e la modellazione dell'utilizzatore finale, aspetto chiave per un'applicazione web moderna. Di seguito saranno analizzati gli strati facenti parte del modello di riferimento [4]:

- *Capabilities*: questo strato racchiude al suo interno tutte le funzionalità offerte dall'applicazione web; tali funzionalità possono essere offerte dal lato server (tipicamente sotto forma di servizi) o in alternativa dal lato client o dall'utilizzatore stesso.

- *Services*: questo strato rende disponibili alla loro fruizione le funzionalità offerte da un'applicazione attraverso un insieme di protocolli e standard esistenti. All'utilizzatore finale di un servizio basta conoscere

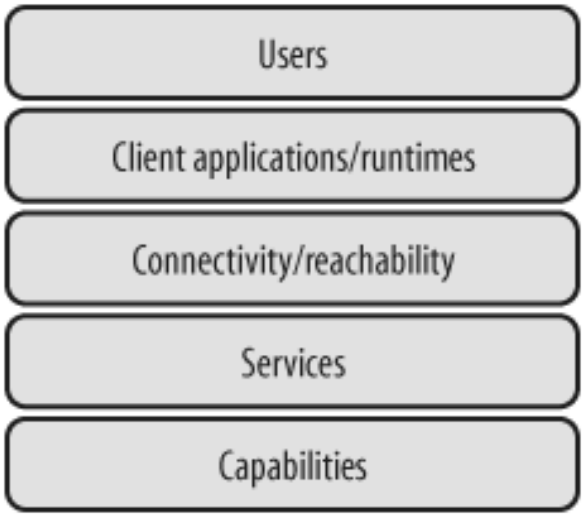


Figura 2.1: Nuovo modello client/server di riferimento

solo il modo con il quale interfacciarsi ad esso, cioè quali sono i dati eventualmente richiesti e prodotti dal servizio, mantenendo trasparente l'aspetto realizzativo. L'architettura tipica situata dietro a tale livello generalmente è la Service Oriented Architecture (SOA), la quale è un elemento fondamentale per internet: grazie a questo paradigma un entità può soddisfare le proprie esigenze utilizzando servizi messi a disposizione da un entità esterna, oppure è possibile realizzare computazione distribuita tra più entità. L'utilizzo della SOA necessita di un sistema di descrizione semantica dei servizi in gioco per permettere a diversi sistemi di interfacciarsi tra loro.

- *Connectivity/Reachability*: questo strato gestisce l'interoperabilità tra sistemi e parti di un sistema attraverso la rete internet; il tutto è permesso grazie ad una serie di protocolli, standard e tecnologie: il protocollo di trasporto di ipertesti HTTP ed il linguaggio di markup degli ipertesti HTML sono ad esempio i componenti fondamentali del web supportati dai browser per permettere l'esecuzione delle applicazioni web. Gli standard legati alla connettività che hanno rivoluzionato il web moderno riguardano i web services e tra quelli più diffusi troviamo AJAX (Asynchronous Javascript and XML), SOAP (Simple Object Ac-

cess Protocol), Web Services-Security (WS-S), Web Services Reliable Exchange (WS-RX) e molti altri; tecnologie come ad esempio AJAX sebbene siano basate su tecnologie già esistenti, hanno dimostrato che unendo le loro forze è stato possibile raggiungere nuovi risultati e in questo caso è stata data una svolta all'aspetto di interattività delle applicazioni web. Oltre ad AJAX, anche i web services hanno contribuito alla crescita del web per quanto riguarda l'aspetto di reachability, in quanto hanno reso possibile la comunicazione di tipo affidabile, dando la possibilità alle parti di sapere quando una comunicazione non è andata a buon fine.

- *Client Applications/Runtime*: questo strato rappresenta la parte client delle applicazioni web e contiene le evoluzioni maggiori nei confronti delle applicazioni web tradizionali, in quanto presenta un proprio strato di runtime e necessita di un'architettura software più vicina ai canoni delle applicazioni desktop che delle applicazioni web tradizionali; le evoluzioni avvenute nelle tecnologie e nei browser hanno permesso la realizzazione di applicazioni web moderne che presentano un elevato livello di interattività e di conseguenza è aumentata la complessità di questo strato concettuale.

- *Users*: l'inclusione di questo strato concettuale è uno dei fattori principali che contraddistinguono una applicazione web moderna da una tradizionale; l'utente ha un ruolo fondamentale all'interno di questo tipo di applicazioni, in quanto è l'attore principale delle interazioni che avvengono al suo interno: un utente può utilizzare una funzionalità dell'applicazione e a sua volta può diventare un fornitore di funzionalità fruite dall'applicazione o da altri utilizzatori. I contenuti generati dagli utenti sono l'entità fondamentale per un applicazione web moderna di successo.

## 2.3 Architettura di riferimento

L'accrescente complessità delle applicazioni web ha reso indispensabile il passaggio dall'architettura classica delle applicazioni web tradizionali ad una nuova architettura che tenesse conto di tutte quelle tecnologie che hanno permesso tale salto di qualità. L'architettura di riferimento non deve tenere

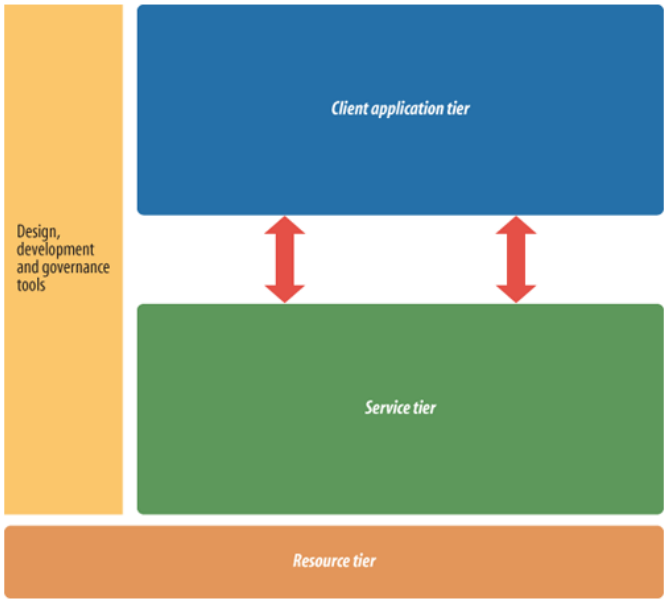


Figura 2.2: Architettura di riferimento ad alto livello

conto degli aspetti implementativi, ma deve solo agevolare la strutturazione di un'applicazione di una certa complessità. Un'architettura di riferimento non deve essere vista come soluzione a tutti i problemi, ma bensì un punto di partenza verso la realizzazione di un problema.

La differenza tra un'architettura e un modello di riferimento è il grado di concretezza con il quale tali artefatti di analisi si rapportano con il risultato finale da produrre: un'architettura contiene aspetti più concreti rispetto ad un modello in modo da essere uno strumento utile allo sviluppatore durante l'implementazione del prodotto finale.

Un'architettura è composta da più blocchi concettuali e può essere rappresentata a più gradi di astrazione: la figura 2.2 sottostante raffigura la nuova architettura di riferimento ad alto livello di astrazione.

I componenti principali di questa architettura sono i seguenti:

- *Risorse*: è lo strato che contiene le funzionalità ed il backend di un sistema che supportano i servizi consumati attraverso internet; tipicamente include databases, files o più in generale risorse dipendenti dal dominio applicativo.

- *Servizi*: è lo strato che utilizza le risorse offerte dallo strato sottostante per offrire servizi che realizzano funzionalità richieste dall'applicazione. Il fruitore dei servizi deve essere a conoscenza dei requisiti di interfacciamento ai servizi per poterne usufruire.

- *Connettività*: è lo strato che rende possibile la fruizione dei servizi offerti da un applicazione. La connettività è gestita da protocolli standard come HTTP e utilizza altrettanti standard per la comunicazione, come ad esempio XML, ma sono esistenti anche altri protocolli e standard altrettanto diffusi.

- *Client*: è lo strato che permette l'interfacciamento dell'utilizzatore dell'applicazione con essa stessa, mettendo a disposizione strumenti che arricchiscono l'esperienza d'utilizzo del software.

- *Progettazione, sviluppo e tools*: è lo strato che racchiude tutti gli strumenti utili allo sviluppatore per realizzare l'applicazione, come ad esempio IDE, strumenti di debugging, ecc.

Ogni strato dell'architettura appena descritta può essere decomposto in sotto-strati aumentando il dettaglio di definizione dell'architettura ed in figura 2.3 è rappresentato ad esempio uno dei possibili livelli di concretizzazione sufficientemente dettagliato per poter essere un buon punto di partenza per lo sviluppo di un'applicazione web. Progettisti e sviluppatori poi possono implementare questa architettura in base alle loro necessità, utilizzando le tecnologie più opportune.

Di seguito saranno analizzati i macro blocchi concettuali mostrando in dettaglio la loro struttura interna.

### Risorse

Questo blocco concettuale contiene tutte le funzionalità core presenti nel sistema che saranno poi esposte attraverso lo strato dei servizi con le tecnologie opportune in base al tipo di fruitori. Alcuni esempi di funzionalità che possono far parte di un'applicazione web moderna sono i sistemi ERP (Enterprise Resource Planning), sistemi CRM (Customer Relationship Management), databases o più in generale risorse persistenti rese disponibili attraverso servizi.

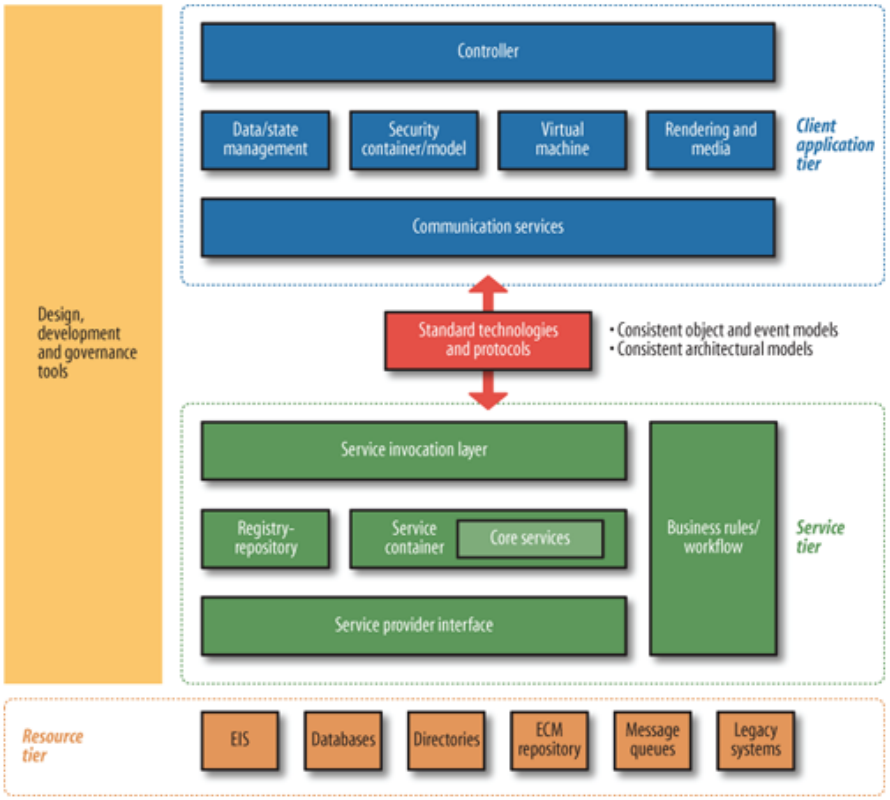


Figura 2.3: Architettura di riferimento in dettaglio

Nei blocchi presentati come risorse nella figura 2.3 ne sono state elencate solo alcune di quelle potenzialmente esistenti:

- *EIS*: con il termine Enterprise Information System si identificano i sistemi di gestione di dati utilizzati all'interno del business applicativo, cioè un componente solitamente presente all'interno di una applicazione web.
- *Databases*: sono tipicamente utilizzati per memorizzare dati in repository strutturate in modo più o meno complesso, dipendente dalla natura e dalla quantità dei dati memorizzati.
- *ECM repository*: con il termine Enterprise Content Management si intende una specializzazione dei sistemi EIS (generalmente realizzato con databases) che si occupa della gestione dei contenuti dell'applicazione web.
- *Message queues*: questo blocco concettuale racchiude al suo interno tutti i meccanismi di comunicazione inter-componente tipicamente realizzati con comunicazioni asincrone.
- *Legacy systems*: quest'ultimo blocco racchiude in se tutti quei componenti ereditati dal passato, cioè ancora utilizzati per diversi motivi, i quali possono essere il frutto di un buon lavoro ancora valido e al passo con le tecnologie e le esigenze, oppure si tratta di componenti che hanno lo scopo di essere rimpiazzati da nuovi.

**Servizi**

Lo strato dei Servizi è suddiviso nei seguenti sotto-blocchi:

- *Service invocation layer*: è lo strato che si occupa del collegamento dei listeners agli eventi di invocazione dei servizi; tipiche realizzazioni di questo strato utilizzano tecnologie come SOAP o XML su protocollo HTTP per generare questi eventi.
- *Service container*: è il core principale del macro blocco dei servizi, il quale si occupa della gestione delle richieste dei servizi e del routing verso i rispettivi componenti che si occupano della loro esecuzione; allo stesso tempo si occupa anche della consegna delle risposte e degli eventuali errori al termine delle esecuzioni dei servizi.

- *Business rules and workflow*: è lo strato contenente i vincoli e le regole legate al flusso di esecuzione dei servizi.
- *Registry/repository*: è lo strato che tiene traccia dell'esecuzione dei servizi memorizzandone dati utili ad esempio per ottimizzare il carico di lavoro.
- *Service provider interface (SPI)*: è lo strato che si occupa della connessione tra lo strato dei servizi e lo strato delle risorse per renderle disponibili alla loro fruizione.

### Applicazione Client

Lo strato Client è suddiviso nei seguenti sotto-blocchi:

- *Controller*: è Il componente principale dello strato di applicazione client e si occupa della gestione degli altri componenti di questo strato. Il concetto di Controller è allineato a quello presente nel pattern Model-View-Controller e quindi si occupa della logica applicativa principale dell'applicazione.
- *Virtual Machine*: lo strato client di un'applicazione può avere più ambienti runtime specifici per ogni tecnologia client-side, ognuno dei quali è emulato nella sua virtual machine.
- *Data/state management*: per offrire una ricca esperienza d'utilizzo all'utente è necessario poter elaborare/memorizzare dati anche lato client e questo strato ne permette la gestione temporanea (in memoria) o permanente (tipicamente databases) se le tecnologie lo permettono.
- *Security container/model*: è lo strato che si occupa di gestire gli aspetti legati alla sicurezza riguardo i possibili scenari che possono intaccare l'incolumità dell'applicazione.
- *Rendering and media*: è lo strato che si occupa della gestione dei processi di rendering e dei media, per presentare all'utilizzatore dell'applicazione l'interfaccia grafica.
- *Communications*: è lo strato che si occupa dell'interfacciamento tra applicazione client e servizi; è vincolato dallo strato di sicurezza ed è orchestrato dal Controller dell'applicazione.

## 2.4 Design Patterns

L'architettura di riferimento illustrata nella sezione precedente può essere realizzata con diversi design patterns a seconda delle esigenze insite nell'applicazione da sviluppare. In questa sezione saranno illustrati a grandi linee alcuni design patterns che forniscono una buona base per un'applicazione web moderna.

#### Service-Oriented Architecture

SOA è un pattern utilizzato per organizzare le funzionalità messe a disposizione da un sistema software sotto diversi domini di proprietà, consentendo le interazioni tra i consumatori delle funzionalità e queste ultime tramite servizi. Il servizio è isolato dal consumatore con una certa opacità che ne nasconde i dettagli interni su come essa è realizzata.

Le architetture SOA permettono l'eliminazione di molte dipendenze tra i sistemi, perché il servizio separa nettamente il consumatore dal fornitore. SOA riduce le duplicazioni delle funzionalità e aumenta il loro isolamento in modo tale da facilitare il processo di testing e di renderne il riutilizzo più agile.

Molti altri pattern dipendono SOA: Mashup e Software as a Service (SaaS) ad esempio, si basano su uno strato di servizi che possono essere mescolati e abbinati per creare nuove applicazioni e potenti esperienze d'utilizzo per gli utenti.

L'accesso non autorizzato e l'abuso di un singolo servizio sono rischi reale che i fornitori di servizi devono prendere in considerazione quando espongono i propri servizi via Internet.

I progettisti devono considerare quali funzionalità sono potenziali candidati per diventare i servizi; se una funzionalità viene utilizzata da un solo altro sistema, può non essere un candidato adatto, invece nel caso opposto, cioè se viene usato da più di un processo, potrebbe essere ideale per realizzare un servizio.

#### Software as a Service

SaaS è un pattern di consegna del software in cui il produttore è responsabile del funzionamento del prodotto fornito ai clienti, i quali possono fruirlo da

diverse postazioni remote. Le funzionalità rese disponibili su internet attraverso servizi software sono utilizzabili direttamente dalle applicazioni web senza dover installare nessun software nella propria macchina.

Utilizzare questo pattern comporta una serie di considerazioni importanti:

- mettere a disposizione la computazione e le risorse ad un ampio numero di utilizzatori, rende complicata la gestione dello scaling dinamico, soprattutto quando avvengono dei picchi di utilizzo;
- questo tipo di soluzione implica l'utilizzo di connettività, senza la quale il software non è disponibile; una soluzione parziale al problema è la gestione della cache locale per un'esecuzione offline delle funzionalità (dove ha senso attuare questo tipo di soluzione).
- il sovraccarico di richieste del software è in alcuni casi un'azione portata avanti da un utente malintenzionato che ha lo scopo di far crollare le prestazioni del software o di addirittura farlo collassare.

### Participation-Collaboration

Questo pattern viene utilizzato quando all'interno dell'applicazione la conoscenza presente è ottenuta attraverso un processo di collaborazione con gli attori del sistema. Lo stesso pattern può essere utilizzato anche per progetti open source, dove gli sviluppatori possono contribuire in questo caso alla crescita del progetto. La particolarità di questo pattern sono i benefici che esso permette all'applicazione di ottenere, ma chi lo utilizza deve fare particolare attenzione alle modalità di gestione del processo di collaborazione, cercando di guidare la comunità di utilizzatori del software al raggiungimento dell'obiettivo comune senza fare apparire un comportamento di dominio troppo restrittivo.

### Asynchronous Particle Update

Questo pattern permette la comunicazione client/server per scambiare messaggi contenenti piccole quantità di dati, sulla base di parametri di comunicazione; utilizzando questo processo di comunicazione è possibile realizzare applicazioni web ricche di interattività, attuando un dinamismo moderno sulle pagine di un'applicazione web che non necessita più di refresh interi di pagine per aggiornare i contenuti presenti al suo interno.

Questo pattern necessita quattro componenti fondamentali:

- la prima parte consiste nella possibilità concessa dal browser di inoltrare un messaggio strutturato ad un indirizzo remoto ed eventualmente di gestire la risposta attesa;
- la seconda parte è il componente lato server che resta in attesa di richieste provenienti dai browser e le elabora quando esse arrivano;
- la terza parte è l'ambiente di runtime presente nel browser che permette la gestione dei dati restituiti come risultato della chiamata inoltrata;
- la quarta ed ultima parte è quella che si occupa dell'aggiornamento dell'interfaccia utente in base all'elaborazione avvenuta dopo il risultato ottenuto durante la comunicazione con il server.

Realizzare un'applicazione web utilizzando questo pattern per migliorarne l'esperienza d'utilizzo può essere un'arma a doppio taglio, perché potrebbe presentarsi il caso in cui per rendere una pagina dinamica sia necessario effettuare molte chiamate lente e si potrebbe raggiungere lo stesso risultato con un refresh della pagina in modo più veloce: l'utilizzo di questo pattern va quindi valutato attentamente in termini di widget AJAX presenti nelle pagine web.

Inoltre l'utilizzo dei servizi deve essere strutturato in modo da tener in considerazione le problematiche legate alla sicurezza e alle performance.

Questo pattern è anche conosciuto con il nome REST (Representational State Transfer) e la tecnologia realizzativa più diffusa è AJAX (Asynchronous JavaScript and XML).

### Rich User Experience

Il processo fondamentale alla base della realizzazione di un'applicazione web che possiede una Rich User Experience (RUE) è l'analisi del comportamento nel mondo reale delle persone fisiche nell'ambito per il quale l'applicazione deve essere creata; questa analisi ha lo scopo di modellare queste interazioni per poterle realizzare al meglio all'interno dell'applicazione web.

Le domande che si deve porre lo sviluppatore per effettuare una buona modellazione hanno lo scopo di rendere l'esperienza di utilizzo dell'applicazione web il più ricca possibile per tutte le tipologie di utilizzatori e soprattutto di renderla personalizzata al punto giusto per il singolo utilizzatore.

Questo pattern implica una maggiore complessità del sistema da realizzare in quanto è presente una parte client molto strutturata e per mantenere una buona indipendenza tra i componenti del sistema, sono utilizzabili i precedenti pattern descritti: ad esempio SaaS, SOA, o i servizi REST sono pattern che permettono una buona separazione tra client e server.

Arricchire l'esperienza di utilizzo però implica anche l'esclusione di una fetta di utilizzatori che non sono in possesso di abilità per utilizzare l'interfaccia dell'applicazione: questo aspetto deve essere tenuto in forte considerazione dagli sviluppatori in base alle esigenze specifiche.

#### Synchronized Web

Questo pattern è utilizzato per modellare le applicazioni web che richiedono tra le loro funzionalità operazioni di sincronizzazione. Tra le operazioni di sincronizzazione troviamo ad esempio le classiche GET e PUT in un modello REST utilizzate per sincronizzare i dati in gioco di un'applicazione, oppure è possibile realizzare ad esempio applicazioni che gestiscono la mancanza di connettività memorizzando dati in locale che saranno sincronizzati attraverso la mediazione con il lato server non appena la connettività sarà tornata a disposizione: la gestione di quest'ultimo tipo di sincronizzazione prevede un particolare trattamento dei dati in modo tale da renderne la sincronizzazione consistente.

Nell'utilizzo di questo pattern è molto importante minimizzare la complessità delle API, utilizzando possibilmente metodi web standard (GET, PUT, ecc.) e generalizzando il più possibile lo sviluppo web.

## 2.5 Model-Driven Engineering

Le applicazioni web moderne hanno rivoluzionato il modo in cui avviene la computazione sul web, partizionandola tra client e server, permettendo un notevole progresso nell'interattività di tali applicazioni; tale rivoluzione è stata riflessa anche nelle metodologie per lo sviluppo web: tutto ciò che è stato consolidato nel passato può essere adattato alle nuove esigenze o bisogna individuare nuove tecniche per lo sviluppo?

Nei precedenti paragrafi sono stati analizzati come sono cambiati i modelli e le architetture di riferimento e sono stati brevemente descritti alcuni tra

i più diffusi design pattern per la progettazione web moderna; in questo paragrafo saranno invece affrontati i cambiamenti legati all'utilizzo del Model-Driven Engineering come approccio per la modellazione, la progettazione e lo sviluppo di un'applicazione web moderna.

Il Model-Driven Engineering (MDE) consiste nella definizione delle funzionalità di un sistema modellandolo in modo platform-independent utilizzando un linguaggio domain-specific (DLS). Tale modello, attraverso un processo di trasformazione model-to-model, potrà essere convertito in un modello platform-specific e attraverso un altro processo di trasformazione model-to-code, sarà generato automaticamente una percentuale importante di codice appartenente ad un linguaggio general purpose (GPL).

L'utilizzo di tale approccio ingegneristico per lo sviluppo di applicazioni web tradizionali è stato consolidato nel corso degli anni, mettendo a disposizione degli sviluppatori dei DSL come ad esempio WebML; nelle applicazioni web moderne l'architettura base è più complessa e sono state introdotte nuove tipologie di funzionalità e nuovi concetti ognuno con la propria semantica: come si porranno le tecnologie esistenti in questo ambito nei confronti di tali cambiamenti?

Il problema può essere risolto con il seguente approccio [5]:

- effettuare una sintesi delle caratteristiche della metodologia MDE utilizzata per lo sviluppo di applicazioni web tradizionali;
- stilare un elenco delle funzionalità/concetti introdotte nelle applicazioni web moderne per valutare estensioni alle notazioni della modellazione di applicazioni web;
- applicare le estensioni al DSL per lo sviluppo di applicazioni web: Web Modeling Language;
- applicare ad un caso di studio le nozioni introdotte;
- effettuare una valutazione finale del DSL ottenuto in termini di espressività, facilità d'uso e implementabilità.

Per le applicazioni web tradizionali esistono diversi DSL, ma tutti quanti hanno in comune le seguenti caratteristiche:

- la modellazione del dominio applicativo avviene utilizzando diagrammi ER, UML, OWL, ecc.

- il front-end dell'applicazione web è rappresentato attraverso un modello ipertestuale che esprime: la composizione dell'interfaccia grafica (pagine web), i contenuti (estratti dagli oggetti del modello del dominio), la navigazione (link tra pagine e utilizzo di form) e la business logic;
- alcuni modelli rappresentano anche la dinamicità del front-end attraverso diagrammi UML delle attività.

### WebML

WebML è un DSL per la modellazione di applicazioni web. Con WebML delle applicazioni web tradizionali ne venivano modellati i seguenti aspetti:

- *Struttura*: Il modello del dominio si definisce attraverso un diagramma ER o un diagramma della classi UML, con il quale si modellano le entità in gioco, i loro attributi, le relazioni e le gerarchie tra di esse.
- *Il modello ipertestuale*: può essere composto da viste multiple, in base alla tipologia di utenti, al device sul quale viene eseguita, ecc. Ogni vista è composta da aree e pagine innestate tra loro; le viste sono partizionate in formers, cioè gruppi di pagine che formano una sezione del sito. Le pagine contengono dei contenuti statici (form, ecc.) o componenti per la pubblicazione dinamica di contenuti (che solitamente provengono dalle entità modellate).
- *Comportamento*: si modellano le operazioni che l'applicazione mette a disposizione dell'utilizzatore (creazione, modifica, cancellazione) che andranno a modificare gli oggetti del modello del dominio.
- *Interazione*: le interazioni sono espresse a livello di pagine o a livello di componente. Le prime definiscono la raggiungibilità di un'area del sito, mentre le seconde sono espresse attraverso link presenti nel sito che corrispondono a diversi tipi di azioni: navigazione, attivazione di un'operazione, passaggio di parametri, ecc.

Per modellare un'applicazione web moderna con WebML dovranno essere apportate al linguaggio delle modifiche che tengono in considerazione i seguenti aspetti:

- *Struttura*: è presente una distribuzione dei dati tra client e server;

- *Comportamento*: anche sul client è presente un application runtime environment, quindi bisogna definire anche la computazione dell'applicazione lato client;

- *Interazione*: la presenza dell'application runtime environment anche lato client apre le porte a diversi tipi di comunicazione (server-client, async events, ecc.) rappresentabili con il modello di dinamicità; anche la possibilità di modificare il contenuto delle pagine senza effettuarne il refresh comporta nuovi aspetti di dinamismo dell'applicazione web.

Lato client è quindi ora possibile memorizzare dati in memoria o in modo persistente, proprio come avviene lato server. L'estensione del modello del dominio richiede solo un arricchimento di due dimensioni: *locazione* (client/server) e *durata* (temporanea/persistente). La distribuzione dei dati comporta però critici problemi, come la trasparenza della locazione, la quale nelle applicazioni web moderne per diversi motivi (solitamente tecnologici) spesso non è realizzabile: durante la fase di modellazione andranno quindi formalizzate le operazioni sui dati che saranno effettuate come politica di allocazione, replicazione e consistenza.

Avendo a disposizione un application runtime environment sia lato server che lato client, si aprono le porte a diversi scenari di distribuzione della computazione, in base alle esigenze dell'applicazione. Le estensioni del DSL dovranno permettere la modellazione di questi nuovi aspetti di computazione senza trascurare l'espressività e la facilità d'uso del DSL stesso. Innanzitutto è necessario distinguere le tipologie di pagine che comporranno l'applicazione, cioè le pagine server (create interamente lato server e renderizzate e gestite lato client), le pagine client (creazione, rendering e gestione lato client) e l'organizzazione della struttura di tali pagine considerando che possono essere innestate tra loro pagine client e server. Dopodiché è necessario definire per ogni componente di una pagina la distribuzione della computazione, in quanto ad esempio i dati potrebbero risiedere dal lato opposto a dove viene eseguita la computazione e quindi non sarebbero disponibili all'interno di tale scope. Nella modellazione di ogni unità di computazione è quindi necessario distinguere quando essa avviene lato server o lato client ed è possibile farlo ad esempio utilizzando delle etichette.

Le interazioni sono modellate con il modello degli eventi, il quale si ottiene estendendo il modello del dominio ed esprimendo ogni evento con un tipo, degli attributi (parametri) e le relazioni con le entità. Gli eventi sono

caratterizzati da un'azione di notifica (send) e una di ricezione (receive). Nel modello ipertestuale si modellano entrambi i tipi di eventi e si collegano con le corrispondenti business logics.

Per quanto riguarda la generazione di codice per le applicazioni web tradizionali veniva generato solo codice per la parte server dell'applicazione; per le nuove applicazioni invece, oltre ad estendere i generatori server side per la gestione dei nuovi tipi di interazione, è necessario aggiungere la generazione di codice per la parte client side.

Le modifiche al linguaggio WebML proposte vanno infine valutate prendendo in considerazione i seguenti aspetti:

- *Espressività del linguaggio*: la distribuzione dei dati è completamente supportata, lasciando alla fase di sviluppo la gestione del problema della consistenza dei dati replicati. Anche l'interattività è ben supportata grazie alla modellazione degli eventi nel modello ipertestuale; questo approccio rende la leggibilità di tale modello più complicata a causa di eventi con diversa natura mischiati insieme nello stesso modello: una possibile soluzione è una separazione in sotto-modelli. Per quanto riguarda la modellazione del comportamento ci sono dei limiti riscontrati ad esempio nel processo di validazione dei form in modalità asincrona e nella gestione degli script lato client.

- *Facilità d'uso e implementabilità*: per ottenere questo tipo di valutazione è necessario confrontare l'utilizzo di questo tipo di sviluppo con altre modalità, come ad esempio lo sviluppo tradizionale o lo sviluppo utilizzando il modello per le tradizionali applicazioni web. Nel primo caso giocano a favore dell'approccio tradizionale l'evoluzione delle tecnologie e delle architetture: in questo periodo storico sono in continua evoluzione e di conseguenza anche l'approccio allo sviluppo model-driven dovrebbe adattarsi ai cambiamenti continuamente. Nel secondo caso invece l'utilizzo di un modello esteso permette di abbattere notevoli costi nello sviluppo di quelle parti non modellate nelle tradizionali applicazioni web, come ad esempio le nuove interazioni client-server, la computazione client side e tutti gli aspetti sottolineati in precedenza. Un aspetto negativo però è stato riscontrato nella modellazione della computazione delle pagine web, in quanto la modellazione della dinamicità della pagine dettata dagli eventi scaturibili e da tutte le loro combinazioni, è molto complessa.

# Capitolo 3

# Evoluzione delle Tecnologie

## 3.1 JavaScript

### 3.1.1 Programmazione "in the large"

Nel capitolo precedente è stato analizzato il cambiamento architetturale avvenuto nel contesto delle applicazioni web ed è emerso che il "pivot" di questa evoluzione sia la parte client, la quale ha comportato di conseguenza una serie di evoluzioni globali architettoniche di una applicazione web.

In passato la programmazione lato client di un'applicazione web era legata a tecnologie che furono concepite con lo scopo di essere funzionali, ma soprattutto semplici; nel primo capitolo sono state analizzate le evoluzioni che hanno portato i primi siti web a diventare quello che oggi sono le moderne applicazioni web e in tali applicazioni durante questo processo di evoluzione, dal punto di vista dell'ingegneria del software l'architettura che si è naturalmente più diffusa è stata quella client/server: tale architettura rispetto alla controparte dei sistemi informatici desktop, a causa dei limiti tecnologici presentava una parte server di complessità molto superiore alla parte client. I recenti passi tecnologici compiuti in avanti nelle tecnologie web client hanno permesso la crescita delle potenzialità anche della parte client di un'applicazione web.

La complessità di un componente software pone da sempre lo sviluppatore di fronte ad uno dei problemi più comuni nel mondo dell'informatica, cioè la strutturazione/organizzazione ottimale del codice prodotto per trarne i

maggiori benefici possibili in termini di riutilizzo del codice, abbattimento dei costi di produzione e manutenzione, leggibilità, ecc.

Mentre la programmazione lato server di un'applicazione web sin dal principio ha permesso allo sviluppatore l'utilizzo di tecnologie di natura già affermata anche per la programmazione in larga scala di applicazioni desktop, per la controparte client tra le tecnologie che per prime hanno permesso la computazione si è diffusa maggiormente JavaScript, un linguaggio di scripting nato inizialmente con molti limiti, ma che nel tempo si è evoluto diventando a tutti gli effetti la miglior soluzione per effettuare computazione lato client.

JavaScript è un linguaggio di scripting interpretato ed è debolmente orientato agli oggetti e debolmente tipizzato. Queste caratteristiche lo hanno portato ad un rapido successo in quanto rispetto a linguaggi ad esempio fortemente orientati agli oggetti e fortemente tipizzati come Java o C/C++ ne condivide la simile sintassi concedendone però un più facile ma meno potente utilizzo.

Per applicazioni web client di contenuta complessità come lo erano le prime che utilizzavano tale linguaggio, JavaScript si è rivelato con il tempo un linguaggio molto apprezzato, soprattutto per la sua natura di linguaggio asincrono estremamente compatibile con le esigenze delle applicazioni web. Le applicazioni web moderne, come è già stato affermato in precedenza, ne fanno della complessità lato client l'arma vincente e l'interrogativo che si pone uno sviluppatore web di fronte a questo cambiamento è se un linguaggio come JavaScript può essere ancora adatto per le nuove esigenze; inoltre tra le tecnologie web più recenti è da sottolineare Node.js, una piattaforma web che nei prossimi capitoli sarà descritta più in dettaglio, la quale si contraddistingue per l'utilizzo lato server di JavaScript: un'applicazione web basata sulla piattaforma Node.js si ritroverebbe potenzialmente scritta completamente in JavaScript, quindi in questo caso le esigenze di avere un linguaggio il più completo possibile sono ancora più accentuate.

Uno dei motivi principali legati al successo di JavaScript è la costante e abbondante partecipazione delle comunità di sviluppatori, la quale, durante il percorso evolutivo di tale linguaggio, ha contribuito alla creazione di numerosi progetti atti a fornire: funzionalità aggiuntive al linguaggio, interi frameworks, librerie, tools e tanto altro materiale utile allo sviluppatore per soddisfare le proprie esigenze.

Nei prossimi paragrafi saranno affrontati gli aspetti più rilevanti legati al linguaggio JavaScript e alla sua evoluzione nel tempo e saranno analizzati

nell'ottica della programmazione "in the large" di applicazioni web moderne; saranno inoltre introdotti nuovi linguaggi (TypeScript e Dart) che si pongono come soluzione alternativa all'utilizzo di JavaScript per questo tipo di applicazioni e nel prossimo capitolo saranno descritti in maggiore dettaglio.

### 3.1.2 Object Oriented Programming

JavaScript è un linguaggio orientato agli oggetti Object-based e non Class-based, cioè gli oggetti sono pure strutture dati che presentano un aspetto strutturale rappresentato da proprietà e un aspetto comportamentale rappresentato da metodi. In un linguaggio Class-based gli oggetti sono basati sul concetto di classe, cioè è necessario prima definire una classe e successivamente si potranno istanziare oggetti di tale classe; in JavaScript la creazione di un oggetto implica invece la creazione di una funzione che corrisponde al costruttore dell'oggetto, all'interno della quale saranno definibili proprietà e metodi.

Nonostante sia assente il concetto di classe, il meccanismo di *ereditarietà* in JavaScript è supportato grazie alla sua natura di linguaggio Prototype-based, ovvero è possibile realizzare l'ereditarietà basandosi sulla clonazione di oggetti detti prototipi. Ogni funzione che definisce un oggetto possiede una proprietà chiamata *prototype* che identifica un oggetto che ha il compito di essere un modello per la clonazione di altri oggetti istanziati utilizzando la funzione costruttore: ogni proprietà e metodo definiti per l'oggetto prototype saranno quindi disponibili per tutti gli oggetti istanziati utilizzando il costruttore che possiede l'oggetto prototype stesso. Di seguito saranno mostrati esempi di codice object oriented per sottolineare le differenze appena descritte tra un linguaggio Class-based e JavaScript (Object-based): saranno anche evidenziate tutte le diverse modalità di realizzazione dell'ereditarietà che è possibile utilizzare con quest'ultimo linguaggio.

#### Linguaggio Class-based

```
//creazione di una classe
class Super
{
    //proprietà
    int num;
```

```
    //metodo
    Super(){
        this.num = 2;
    }
}

//istanziare un oggetto
int obj = new Sub();

//ereditarietà
class Sub extends Super
{
    int num2;
    Sub(){
        super();    //costruttore superclasse
        this.num2 = 4;
    }
}
```

**JavaScript: Pseudo-classical Inheritance**

```
function Super () {
    this.num = 2;
}

function Sub() {
    this.num2 = 4;
}
Sub.prototype = new Super();

var obj = new Sub();
```

**JavaScript: Constructor Chaining**

```
function Super () {
    this.num = 2;
}

function Sub() {
    Super.apply(this, arguments);
    this.num2 = 4;
}

var obj = new Sub();
```

### JavaScript: Parasitic Inheritance or Power Constructors

```
function createSuper() {
    var obj = {
      num: 2
  };

  return obj;
}

function createSub() {
    var obj = createSuper();
    obj.num2 = 4;
    return obj;
}

var obj = createSub();
```

### ECMAScript 5th Ed. Object.create method

```
//Controllo su esistenza di implementazione nativa del metodo
if (typeof Object.create !== 'function') {
    Object.create = function (obj) {
        //costruttore vuoto
        function F() {}
```

```
        //setta come prototype l'oggetto base obj
        F.prototype = obj;
        //ritorna l'oggetto creato con il giusto [[Prototype]]
        return new F();
  };
}

var superInstance = {
    num: 2
};

var obj = Object.create(superInstance);
obj.num2 = 4;
```

Gli esempi di codice illustrati mostrano come creare un oggetto che ha le caratteristiche (proprietà e metodi) ereditate da un altro tipo di oggetto e che le estende con nuove caratteristiche. La differenza tra i primi due tipi di ereditarietà che è possibile realizzare in JavaScript consiste nella modalità di fruizione delle caratteristiche ereditate, cioè nel primo caso per l'accesso alla proprietà num, bisogna passare dall'oggetto prototype (*obj.prototype.num*), mentre nel secondo caso, utilizzando il metodo apply, le proprietà sono accessibili direttamente dall'oggetto (*obj.num*) Il terzo caso invece consiste nell'utilizzare il meccanismo "object augmenting", il quale non necessità l'utilizzo dell'operatore *new* per l'istanziazione di un nuovo oggetto. L'ultima tecnica per realizzare l'ereditarietà è quella proposta da Douglas Crockford che consiste nel copiare l'oggetto prototype dell'oggetto ereditato nell'oggetto prototype dell'oggetto ereditario.

L'ereditarietà è uno strumento molto importante nella programmazione orientata agli oggetti, in quanto permette una strutturazione più avanzata del codice, ma soprattutto permette la riusabilità, un aspetto che consente un notevole abbattimento dei costi durante lo sviluppo software. JavaScript rispetto ad un linguaggio Class-based permette l'ereditarietà in modo più flessibile, in quanto è possibile estendere/modificare un oggetto in qualsiasi momento durante il suo ciclo di vita, mentre in un linguaggio Class-based un oggetto rimane per tutto il suo ciclo di vita della stessa natura definita dalla classe che lo modella. Questa possibilità può essere considerata un vantaggio, ma ne è necessario un accorto utilizzo per non perdere il control-

lo dell'orientamento agli oggetti del codice prodotto, soprattutto quando si trattano applicazioni su larga scala.

### 3.1.3 Tipizzazione dinamica

JavaScript è linguaggio debolmente tipizzato, cioè alle variabili non sono associati dei tipi di dato, ma solo dei valori, che possono dinamicamente cambiare tipo durante il ciclo di vita della variabile. Ad esempio ad una variabile può essere inizialmente associato un valore numerico per poi successivamente essere trasformato in stringa. La tipizzazione dinamica consente lo stile di tipizzazione chiamato *Duck Typing*, il quale consiste nella possibilità di determinare la semantica di un oggetto in base ai metodi ed alle proprietà che esso possiede e non invece in base al suo tipo di dato. La gestione di eventuali errori legati all'utilizzo improprio dei valori associati alle variabili avviene a runtime in quanto JavaScript è un linguaggio interpretato: questo aspetto implica vantaggi dal punto di vista della flessibilità durante la programmazione, ma anche svantaggi legati invece ad aspetti di performance e di controllo di errori (sicurezza), in quanto i controlli vengono fatti a runtime subito prima dell'esecuzione delle istruzioni da eseguire; in un linguaggio compilato di conseguenza si invertono i vantaggi e gli svantaggi.

Per quanto riguarda la realizzazione di applicazioni web su larga scala, un linguaggio interpretato rende la gestione di errori un passaggio molto delicato da affrontare durante lo sviluppo in quanto su una mole di codice molto sostanziosa, un eventuale errore di type checking individuato a runtime può comportare una notevole perdita di tempo: l'errore spesso è difficile da individuare subito e tutto il codice prodotto/modificato durante il periodo intervallato dalla creazione dell'errore e dalla sua individuazione potrebbe essere compromesso e di conseguenza da ristrutturare. In un linguaggio compilato grazie al type checking effettuato a compile type questi errori vengono individuati subito, con un conseguente vantaggi in termini di tempo di lavoro.

### 3.1.4 Programmazione asincrona

La programmazione con i classici linguaggi non web è tipicamente sincrona, cioè con operazioni di I/O bloccanti. L'ottimizzazione delle performance con questi linguaggi si ottiene mediante l'utilizzo dei Threads, permettendo così l'esecuzione di altre porzioni di codice mentre si è in attesa della terminazione

di tali operazioni. Tale approccio implica quindi la presenza di Threads in stato di attesa ogni qualvolta si incontri un'operazione di I/O sincrona. JavaScript è un linguaggio basato invece sulla programmazione asincrona, cioè con operazioni di I/O non bloccanti.

### Callback

Per realizzare I/O in modo asincrono si utilizzano funzioni anonime chiamate *Callback*, le quali vengono eseguite tipicamente al termine dell'operazione effettuata: questo metodo implica una serie di problematiche che lo sviluppatore deve conoscere bene per evitare di incorrere in errori tipici di questo tipo di programmazione; programmare con le Callback comporta:

- un continuo salto tra porzioni di codice soprattutto nel momento in cui vengono eseguite molte I/O;
- in caso di operazione di input, tutta la gestione di tale dato richiesto va inserita all'interno della Callback;
- nel caso debba essere eseguito del codice in successione alla chiamata di un operazione di I/O, tale codice entra in concorrenza con il codice presente all'interno della Callback e quindi non è garantito lo stesso flusso di esecuzione degli statements aprendo così diversi scenari in base alle scelte di schedulazione effettuate dal Sistema Operativo;
- gestire I/O asincrone sequenzialmente e parallelamente (soprattutto) è complesso ed eventuali errori sono difficili da gestire.

Esistono molte librerie che agevolano la gestione della programmazione asincrona mediante utilizzo di Callback, ma esistono a loro volta anche altre modalità di programmazione asincrona che aggirano i problemi legati all'utilizzo delle Callback.

### Promise

In fronte all'esecuzione di un operazione di I/O con il classico meccanismo che prevede l'utilizzo di Callback, tale funzione di gestione della fase di post processing dell'operazione viene passata come parametro all'atto dell'invocazione dell'operazione di I/O stessa. Una soluzione alternativa prevede l'utilizzo di

un concetto chiamato *Promise*: l'operazione di I/O quando viene eseguita restituisce subito un oggetto chiamato "promise", il quale in seguito potrà essere utilizzato per gestire il risultato dell'I/O. Questa tecnica garantisce l'asincronismo puro nei casi di codice eseguito in successione alla chiamata dell'I/O, ottenendo così un unico scenario esecutivo. È una soluzione più elegante in termini di leggibilità del codice e risolve tutti i difetti evidenziati nell'utilizzo delle Callback.

### Coroutine

Un'ulteriore soluzione alternativa prevede l'utilizzo della funzionalità *Coroutine* introdotta nello standard ECMAScript versione 6: tale versione sarà ufficialmente presente nei comuni Browser solo fra qualche anno, ma nel frattempo le Coroutine sono già presenti in alcuni linguaggi che compilano in JavaScript il codice prodotto. Questa funzionalità permette allo sviluppatore di realizzare operazioni sincrone bloccandosi in attesa del suo completamento.

### Vantaggi e Svantaggi

In un'applicazione web moderna è presente una forte componente di interazione tra client e server che di conseguenza comporta un elevato utilizzo di programmazione asincrona. I pattern brevemente descritti in precedenza sono supportati da numerose librerie che aiutano quindi lo sviluppatore a gestire i casi critici di questo tipo di programmazione. La programmazione asincrona comporta un grande vantaggio in termini di performance soprattutto quando si devono effettuare operazioni di I/O che impiegano molto tempo: il vantaggio deriva dal fatto che durante tale operazione il controllo del flusso del programma principale può avanzare e al termine dell'esecuzione dell'operazione, può essere eseguita una Callback di post processing. Tale vantaggio deve però essere ben gestito dallo sviluppatore, perché come è stato descritto in precedenza, questo tipo di programmazione presenta diversi aspetti critici. La conseguenza naturale del precedente vantaggio è quindi la difficoltà di utilizzo di questo paradigma di controllo del flusso, il quale rispetto al classico paradigma sequenziale soffre il non naturale susseguirsi delle istruzioni da eseguire.

### 3.1.5 JavaScript lato server

JavaScript al momento della sua nascita fu proposto come linguaggio di scripting sia per la programmazione lato client che lato server, ma a causa di diversi motivi (scarse performance, librerie scarna, concorrenza migliore - Java, ecc.) la programmazione server side non ebbe successo, mentre per la programmazione lato client ebbe la meglio sulle tecnologie esistenti (Applet Java, ecc.).

Con il passare degli anni JavaScript ha subito un processo di evoluzione che lo ha riportato alla ribalta anche come linguaggio per la programmazione lato server: sono migliorate notevolmente le performance di esecuzione del linguaggio con il motore V8 ed è nato un mondo di librerie e framework che toccano quasi qualsiasi aspetto legato ad una applicazione web.

Sono nate quindi tecnologie che portano JavaScript lato server come ad esempio *Node.js* e questa volta si pongono come valide alternative alle tecnologie concorrenti. Rispetto a tali tecnologie JavaScript si contraddistingue soprattutto per la sua natura di linguaggio asincrono, che in ambito server side incide positivamente sull'aspetto di scaling del web server: con un linguaggio asincrono tale aspetto è gestito seguendo il modello Event-based, mentre con un linguaggio sequenziale si segue il modello Thread-based; questi due differenti modelli saranno confrontati nel prossimo paragrafo.

Un altro vantaggio di avere JavaScript lato server è che lo sviluppo di un'applicazione web può essere realizzato completamente con lo stesso linguaggio: questo vantaggio è la fonte principale dei pensieri che si stanno velocemente diffondendo riguardo ad un possibile futuro predominio di JavaScript come linguaggio globale per la programmazione web; allo stesso tempo essendo JavaScript anche un linguaggio molto criticato, stanno nascendo progetti che propongono nuove tecnologie che hanno lo scopo di sostituire JavaScript in questo processo di "conquista" del predominio nella programmazione web.

### 3.1.6 Modelli Thread-based e Event-based: differenze

Di seguito saranno descritti i due modelli di gestione dello scaling di un web server:

- *Thread-based Model* (Multithreading): in fronte alla ricezione di una connessione da parte di un client, il web server resta occupato durante tutta la sua gestione, in quanto anche eventuali operazioni di I/O sono

bloccanti. Per scalare in modo efficiente le prestazioni di un web server basato su un linguaggio sequenziale, bisogna ricorrere alla programmazione multi-thread. In fronte ad ogni operazione di I/O bloccante è necessario creare nuovi thread per continuare a gestire le successive richieste. Inoltre per ottimizzare la parallelizzazione delle richieste che giungono dai client, è necessario gestirle ognuna su un thread dedicato: questa scelta obbligata comporta un elevato context-switching tra thread che aumenta esponenzialmente con il crescere del numero di connessioni.

- *Event-based Model* (Asyncronous Event-driven): la presenza di un operazione di I/O nel modello di programmazione asincrona non blocca l'esecuzione del web server, che avanza quindi con il proprio lavoro da svolgere; al termine dell'operazione di I/O, l'handler di gestione della sua fase di post processing viene accodato con i blocchi di codice in attesa di essere processati: quando esso raggiunge la testa della coda viene eseguito. Questo metodo di scaling del web server è molto efficiente in quanto non ci sono attese bloccanti in fronte a operazioni di I/O e il meccanismo di gestione asincrono ne permette una gestione ottimizzata a livello di strutturazione del codice, senza bisogno di provvedere ad una gestione complessa come quella thread-based.

### 3.1.7 MV* design patterns

Il pattern Model-View-Control è il più utilizzato per le applicazioni incentrate sulla presentazione di contenuti, nelle quali sono ben individuabili e differenziati tra loro i tre componenti fondamentali sui quali si basa questo pattern. In una applicazione web moderna basata interamente su JavaScript, l'utilizzo del pattern MVC basilare non è sufficientemente espressivo in termini di modellazione e distinzione di ogni aspetto presente nell'applicazione. [6]

Di seguito saranno descritti brevemente alcuni pattern basati su MVC dei quali se ne trovano alcune implementazioni proposte da linguaggi non isomorfi come JavaScript e sarà poi illustrato un nuovo pattern chiamato *Resource-View-Presenter* mostrando come esso possa essere più idoneo per la programmazione completamente in JavaScript di una applicazione web.

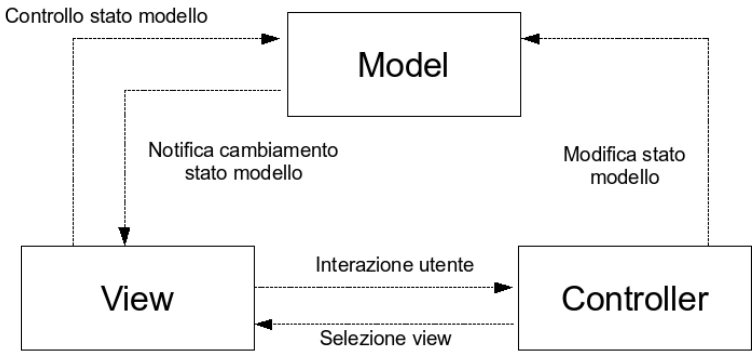


Figura 3.1: Pattern MVC classico

### MVC classico

L'MVC separa in modo distinto le componenti di vista (view), di controllo (controller) e di modello (model) di un applicazione, mantenendo tra loro le corrette relazioni esistenti.

- *model*: fornisce i metodi per accedere ai dati utili all'applicazione;
- *view*: visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti;
- *controller* riceve i comandi dell'utente (in genere attraverso il view) e li attua modificando lo stato degli altri due componenti.

Questo pattern è di uso comune nelle applicazioni incentrate particolarmente sull'interattività con l'utente, il quale attraverso l'utilizzo delle viste interagisce con le funzionalità applicative e con le risorse in essa contenute. Il front-end delle applicazioni web moderne è un classico esempio di compatibilità con l'MVC. Volendo modellare invece interamente (includendo lo strato server side) un'applicazione web con questo pattern, esso potrebbe non essere la soluzione migliore, in quanto lo stretto legame tra view e model necessita una gestione indiretta quando le view (lato client) controllano lo stato di model presenti lato server.

### MVP and MVVM

Sia il *Model-View-Presenter* che il *Model-View-ViewModel* sono simili all'MVC, ma si differenziano per i seguenti aspetti:

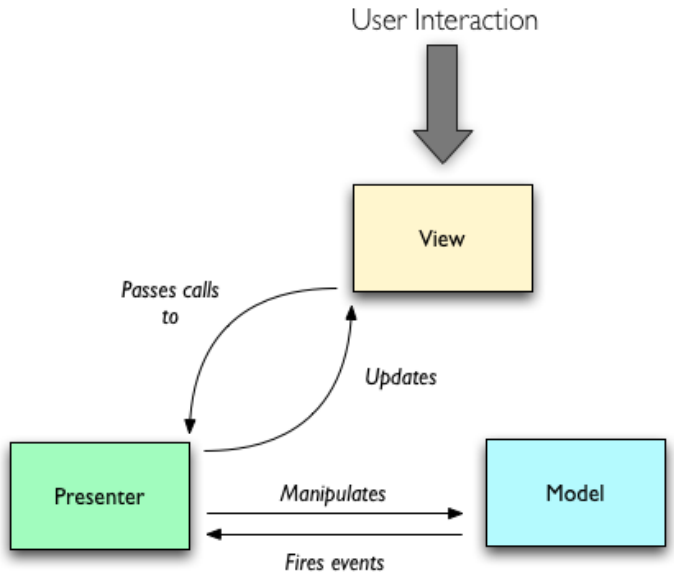


Figura 3.2: Pattern Model-View-Presenter

- Le viste non hanno un riferimento diretto con i modelli;
- I Presenter (o ViewModel) hanno un riferimento alle viste e gli aggiornamenti si basano sui cambiamenti del modello.

MVP e MVVM si distinguono tra loro solo per il fatto che nel MVVM gli aggiornamenti che avvengono nel ModelView si riflettono nelle View grazie ad un robusto motore di associazione dati. L'aspetto positivo di questi pattern è che sono altamente compatibili con l'unit-testing in quanto lo stato delle view è per definizione rispettivamente contenuto all'interno di metodi chiamati dal Presenter nel caso del MVP e all'interno di proprità settabili dal ModelView nel MVVM.

Anche queste varianti del MVC sono perfettamente compatibili con lo sviluppo del front-end di un applicazione web, in quanto aggiungendo un livello di routing è possibile passare il controllo all'appropriato Presenter (o ViewModel) il quale a sua volta aggiorna la view corrispondente e rimane in ascolto di eventi da essa scaturiti. Essendo disaccoppiati i modelli dalle viste, questi pattern, con l'aggiunta di uno strato che si occupa della comu-

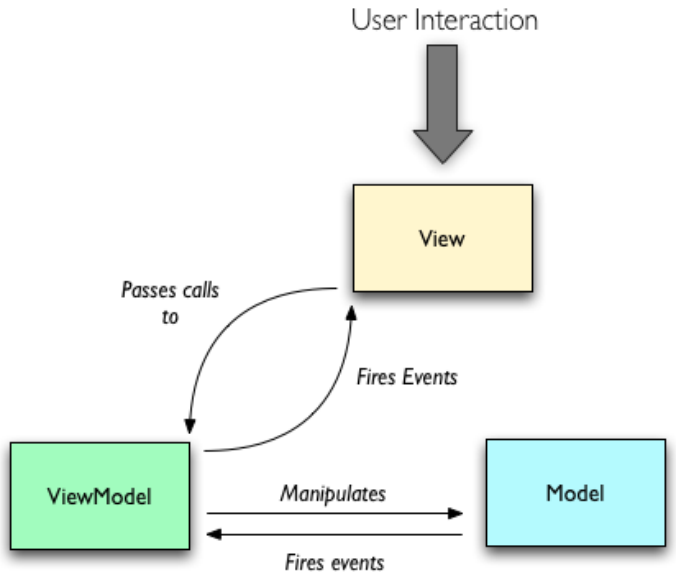


Figura 3.3: Pattern Mode-View-ViewModel

nicazione client-server, si rendono idonei anche all'utilizzo nel lato server di un'applicazione web.

### Real-Time

Tra le esigenze che stanno emergendo nelle moderne applicazioni web, trova un ruolo importante l'aspetto legato alle funzionalità real-time, le quali permettono una comunicazione bidirezionale tra client e server. Queste funzionalità comportano un importante cambiamento nella modellazione del software, in quanto sul lato server è necessario modellare tutti gli aspetti che esse implicano.

I pattern visti in precedenza non sono concepiti però per la modellazione di viste statiche come quelle presenti lato server e quindi si rende necessario l'utilizzo di un pattern, che attraverso l'applicazione di alcune modifiche nei confronti di questi pattern analizzati, renda possibile la modellazione in modo globale di un applicazione web.

**Resource-View-Presenter**

I principali aspetti legati alla nascita di questo pattern sono i seguenti:

- necessità di disaccoppiamento tra Model e View (per permettere sia l'esistenza di View temporanee - generate server side - che persistenti - manipolate client-side);
- modellazione e pianificazione distinta tra client e server;
- spostare la business logic sul Model piuttosto che sul Presenter (mantenendo su di essa solamente gli aspetti legati alle View e allo stato globale dell'applicazione);
- necessità di View più leggere a discapito di Presenter più pesanti (per renderle consistenti con le moderne tecnologie di templating);
- Presenter e Model sono persistenti (consentendo così la creazione di funzionalità real-time).

L'implementazione di questo pattern differisce dal lato client a quello server per la tipologia di viste che essi presentano, in quanto lato server saranno presenti solamente viste temporanee non interfacciate con i Presenter. Nel caso di implementazione web services, le view sono addirittura praticamente inesistenti.

La particolarità che contraddistingue questo pattern dagli altri visti in precedenza è la sua predisposizione alle funzionalità real-time che abilita lo sviluppatore a focalizzare l'applicazione sulla sua business logic, piuttosto che sul suo sottostante strato di trasporto delle comunicazioni.

L'utilizzo di questo pattern agevola notevolmente anche gli aspetti di incapsulamento e riutilizzo del codice, aspetti importantissimi soprattutto nello sviluppo di applicazioni di grandi dimensioni.

### 3.1.8 Futuro di JavaScript

JavaScript è basato sul linguaggio standard ECMAScript, il quale nel corso degli anni ad ogni suo passo di evoluzione ha prodotto una sua nuova versione; ogni browser a sua volta in fronte ad una nuova versione di ECMAScript ha dovuto produrre un suo nuovo aggiornamento per essere compatibile con

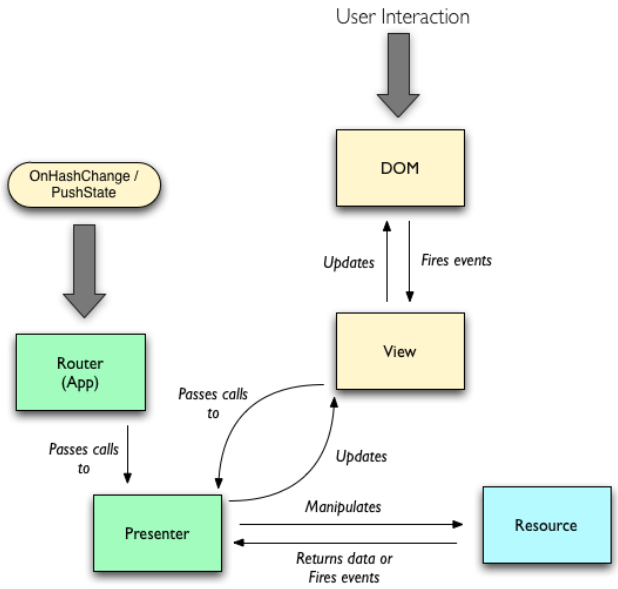


Figura 3.4: Client-side Resource-View-Presenter

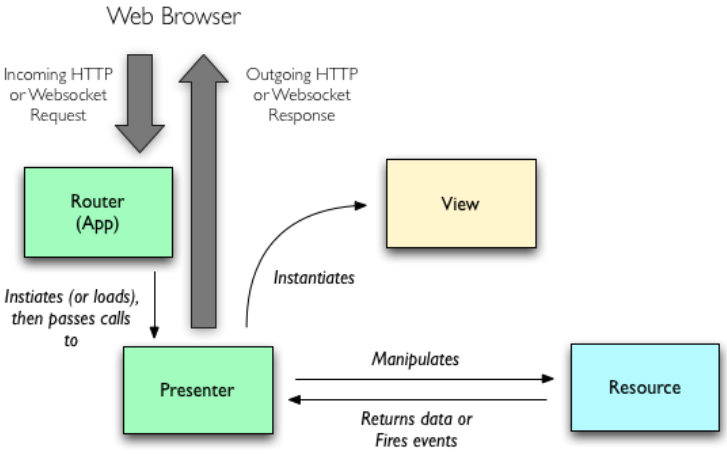


Figura 3.5: Server-side Resource-View-Presenter

le applicazioni web scritte con tale linguaggio. La conseguenza è un'elevata frammentazione tra le versioni dei browser utilizzate dagli utenti; inoltre le applicazioni web che hanno subito variazioni nel tempo o che utilizzano librerie di terze parti, si ritrovano ad essere scritte con varie versioni di JavaScript stesso. Gli aspetti appena sottolineati influiscono anche sugli aggiornamenti stessi del linguaggio ECMAScript, in quanto esso con le modifiche apportate deve garantire comunque il funzionamento di codice scritto con le versioni precedenti.

La versione ECMAScript 6 apporterà nuove funzionalità e caratteristiche al linguaggio, ma prima che essa potrà essere utilizzabile dagli sviluppatori dovrà essere resa compatibile anche dai principali browser. Le specifiche di tale aggiornamento sono però già state diffuse ed è possibile già utilizzarle nello sviluppo di applicazioni web grazie a tecnologie basate sul concetto di compilazione in versioni precedenti di ECMAScript, cioè linguaggi che sfruttano le nuove caratteristiche dell'ECMAScript 6 che però prevedono un atto di compilazione che produce codice ECMAScript di versioni precedenti compatibili con il maggior numero possibile di browser: si va quindi verso il concepimento di JavaScript come linguaggio assembly per il web.

Le novità apportate dall'ultima versione di ECMAScript miglioreranno notevolmente la qualità e le potenzialità del linguaggio JavaScript, ma a causa della critica legata a tale linguaggio in quanto ritenuto da molti non solido, stanno nascendo progetti di diversa natura che hanno lo scopo di guadagnare la leadership nella programmazione web per il futuro: tra questi progetti i due che hanno riscosso maggior interesse sono Dart e TypeScript proposti relativamente da Google e Microsoft.

### TypeScript e Dart

Sia TypeScript che Dart si propongono come soluzione alternativa per lo sviluppo di applicazioni web di elevate dimensioni, in quanto presentano caratteristiche presenti nella versione 6 di ECMAScript ancora assenti in JavaScript che offrono potenzialità aggiuntive allo sviluppatore. La differenza sostanziale tra i due linguaggi è la loro natura, in quanto Dart è un linguaggio indipendente da JavaScript, che ha l'obiettivo di sostituirlo grazie anche all'esistenza di una propria Virtual Machine sulla quale esso sarà eseguito: tale VM garantisce performance migliori delle VM per JavaScript presenti nei browser più diffusi; TypeScript invece si propone come superset di JavaScript

in quanto ne condivide completamente tutte le caratteristiche aggiungendone però anche delle nuove. Entrambi i linguaggi mediante il processo di compilazione trasformano il codice in puro JavaScript risolvendo così il problema della compatibilità con i browser più diffusi. Nei prossimi capitoli saranno analizzati in dettaglio tutti gli aspetti legati a queste due tecnologie.

Il differente approccio utilizzato per questi due progetti pone però TypeScript in vantaggio nei confronti della concorrenza, in quanto il processo di apprendimento di tale linguaggio è molto facilitato grazie alla piena retro-compatibilità con JavaScript e soprattutto alla conseguente vasta disponibilità di numerose librerie legate a quest'ultimo. Dart invece si pone come nuovo linguaggio che prevede la classica fase di apprendimento iniziale e, non essendo compatibile con JavaScript non permette l'utilizzo delle librerie ad esso legate. Il vantaggio di offrire performance superiori in caso di utilizzo puro del linguaggio eseguito sulla VM creata da Google, è confinato attualmente all'utilizzo del solo browser Chrome e, finché tale VM non sarà diffusa anche sugli altri browser principali, questo vantaggio non potrà essere sfruttato dalle applicazioni web sviluppate in Dart.

Nel caso in cui Dart riesca in futuro ad ottenere il successo sperato, si potrebbe concretizzare la fine del linguaggio JavaScript, mentre nel caso di TypeScript un eventuale suo successo non comporterebbe cambiamenti a livello di vitalità di JavaScript, in quanto rimarrebbe comunque utilizzato come linguaggio assembly e rimarrebbero aperte le porte per una possibile ribalta futura quando esso offrirà la compatibilità con lo standard ECMAScript 6.

### Nuove piattaforme

L'utilizzo di JavaScript negli ultimi tempi si sta diffondendo sempre di più a tal punto da essere utilizzato anche per realizzare applicazioni eseguibili su piattaforme diverse da quella web. Abbiamo già anticipato il caso di Node.js al quale se ne affiancano altri già esistenti da tempo e altri ancora che nasceranno in futuro.

Con l'avvento dei moderni dispositivi mobile quali smartphone, tablet, netbook, ecc., lo sviluppo di applicazioni Web ha aperto le porte anche a nuovi ambienti dotati ognuno di una piattaforma specifica. Le prime piattaforme mobile commercializzate seguirono dal punto di vista dello sviluppo software le orme dei sistemi desktop, cioè esse permettevano l'esecuzione di applicazioni native sviluppate con uno specifico linguaggio, fornendo al sup-

porto dello sviluppatore nella maggior parte dei casi anche un ricco ambiente composto da SDK e IDE. Tra le attuali principali piattaforme di successo è possibile citare Android, iOS, BlackBerry e WindowsMobile, le quali permettono la realizzazione di applicazioni native in linguaggi come Java, ObjectiveC, C, C++ e C#

Tali piattaforme mobile tra le applicazioni native possono presentare anche browser proprio come avviene per le piattaforme desktop e, attraverso tali applicativi, permettono l'esecuzione di applicazioni web con ovvie conseguenze legate alle differenze hardware presenti tra gli ambienti desktop e mobile. Lo sviluppo di applicazioni Web con l'avvento di queste moderne piattaforme ha dato luogo ad una nuova tipologia di applicazioni, cioè le cosiddette *Applicazioni Mobile Web*.

Le applicazioni Web mobile non apportano però novità a livello di utilizzo delle tecnologie Web, in quanto oltre al diverso target di dispositivi sui quali saranno eseguite le applicazioni, non ci sono cambiamenti a livello di architetture e pattern di utilizzo.

Il primo passo verso un nuovo utilizzo delle tecnologie Web l'ha compiuto *PhoneGap*, una tecnologia in grado di generare codice nativo per alcune piattaforme moderne di successo a partire da codice sorgente sviluppato proprio con le tecnologie Web. Tali piattaforme, come è stato anticipato in precedenza, per lo sviluppo di applicazioni native propongono ognuna una tecnologia standard che meglio sposa gli aspetti presenti su tali piattaforme: PhoneGap offre quindi un livello di astrazione superiore che permette di sviluppare applicazioni con tecnologie web in modo platform-independent. Il risultato dell'utilizzo di questa tecnologia è un notevole guadagno in termini di durata del ciclo di realizzazione del software e di conseguenza un guadagno anche in termine di denaro; il tutto però a discapito delle performance in quanto l'esecuzione di codice JavaScript su tali piattaforme, non essendo fornito un supporto nativo al linguaggio, richiede l'utilizzo di componenti aggiuntivi che rendono così le performance finali nettamente peggiori rispetto allo sviluppo con linguaggio nativo.

Nonostante il mercato delle piattaforme mobile e desktop sia ben delineato da piattaforme vincenti e moderne, nuovi progetti stanno nascendo orientando l'innovazione sullo sviluppo del software che in esse dovrà essere eseguito. L'obiettivo in comune di alcuni di questi nuovi progetti è quello di aprire il più possibile le porte allo sviluppo del software senza vincolare gli sviluppatori all'utilizzo di una particolare tecnologia. Grazie all'evoluzione

del Web, le tecnologie che ruotano attorno ad esso hanno avuto negli ultimi anni oltre ad una crescita in termini di potenza e qualità, una crescita esponenziale della loro diffusione; in aggiunta si tratta di tecnologie open source che seguono degli standard in continua evoluzione e per tutti questi motivi questi nuovi progetti hanno in comune l'obiettivo di offrire la compatibilità con queste tecnologie a livello di piattaforma. Tra questi progetti è possibile citare Tizen, Firefox OS, Ubuntu, MeeGo, BlackBerry 10 e Windows 8.

Su queste nuove piattaforme sarà quindi possibile eseguire applicazioni web rispettando eventuali vincoli imposti e utilizzando eventuali SDK fornite: di conseguenza potranno essere facilmente utilizzate applicazioni già esistenti con l'eventuale aggiunta di customizzazioni minime dettate da possibili vincoli imposti.

## 3.2 Concorrenza

In questi primi capitoli è stata analizzata l'evoluzione della piattaforma web e delle tecnologie annesse e si è potuto apprezzare come essa sia passato da essere una repository di documenti ad una piattaforma applicativa distribuita. Le applicazioni web moderne hanno cancellato il gap tecnologico con le applicazioni desktop ed hanno aggiunto vantaggi in termini di maggior portabilità del software, facilitazione del processo di deployment e di manutenzione. I browser sono anch'essi evoluti e le applicazioni web sono diventate sempre più ambiziose cercando di sfruttare al massimo le risorse disponibili (i games sono tra le applicazioni che sfruttano maggiormente le risorse e le tecnologie): le risorse sono quindi il fattore più importante da considerare quando si analizzano le funzionalità di un'applicazione. Scalare un'applicazione è una processo complesso, soprattutto quando si devono prendere decisioni sull'architettura e sulle tecnologie da utilizzare. Inoltre, rispetto alla piattaforma desktop, l'esecuzione dell'applicazione è dipendente da fattori aggiuntivi come la fluttuazione dinamica delle richieste provenienti dai client e delle risorse disponibili, le numerose combinazioni di piattaforme esecutive, ecc. Sfruttare la programmazione concorrente multi-core lato client è ora possibile grazie ai web workers introdotti con HTML5, i quali sono ispirati al *modello ad attori*.

Più in generale realizzare sistemi concorrenti apporta i seguenti benefici al software prodotto:

- *Performance*: rispetto ad un sistema realizzato mediante programmazione sequenziale, un sistema concorrente astrae dall'hardware presente nella macchina sulla quale viene eseguito il software e ne permette l'esecuzione in modo potenzialmente parallelo, in quanto in presenza di architetture single-core l'esecuzione delle istruzioni avviene comunque in modo sequenziale, mentre in presenza di architetture multi-core l'esecuzione delle istruzioni avviene in modo parallelo. La gestione della concorrenza in architetture single core comporta però per certi tipi di parallelismi dei peggioramenti delle performance, in quanto l'esecuzione delle istruzioni avvenendo in modo comunque sequenziale deve subire il grado di computazione aggiuntiva necessaria per gestire la programmazione concorrente: paragonando questo minimale degrado di performance con il notevole guadagno ottenuto in caso di esecuzione su architetture multi-core, si può concludere semplificando che la realizzazione di sistemi concorrenti comporta vantaggi positivi in termini di miglioramento delle performance.

- *Responsiveness*: grazie alla programmazione concorrente è possibile disaccoppiare l'esecuzione del codice che si occupa degli eventi scatenati dall'interfaccia grafica di un'applicazione dall'interfaccia grafica stessa, in modo tale da rendere sempre reattivi i componenti presenti nell'interfaccia e di eseguire tali operazioni (in alcuni casi anche molto onerose in termini di computazione) su thread separati; il risultato è che le interfacce grafiche sono sempre reattive alle interazioni dell'utente senza mai bloccarsi.

- *Robustezza*: la programmazione concorrente permette anche di architetturare il codice di un applicazione in modo tale da non soffrire dei possibili blocchi avvenuti in un componente; sfruttando ad esempio meccanismi di messaggistica asincrona è possibile anche ripristinare eventuali componenti bloccati.

- *Distribuzione della computazione*: un sistema concorrente può essere anche eseguito sfruttando l'hardware di più macchine contemporaneamente realizzando così software dalle performance scalabili.

### 3.2.1 Modello ad Attori

Il modello ad attori è basato appunto sul concetto degli attori, entità autosufficienti, indipendenti ed interattive, dalle quali un sistema è composto e che attraverso scambio di messaggi asincrono permettono la realizzazione di computazione concorrente.

Un attore deve avere le seguenti caratteristiche:

- interattività tra attori (attraverso ricezione/invio di messaggi);
- comunicazioni asincrone;
- possiede uno stato locale incapsulato caratterizzato da proprietà e comportamento e modificabile solo attraverso scambio di messaggi;
- esecuzione dei metodi (comportamento) atomica;
- comportamenti di attori eseguiti in concorrenza tra loro;
- un attore può creare un altro attore;
- un attore è identificato da un nome univoco.

Il modello ad attori garantisce le seguenti proprietà:

- *Fairness*: sia nella schedulazione degli attori che dei messaggi.
- *Location trasparency*: il codice sorgente relativo ad un attore non ha dipendenze dal luogo nel quale viene eseguito.
- *Migration trasparency*: gli attori possono essere spostati sulla rete computazionale in base alle necessità.

#### Ciclo di vita di un attore

Ad ogni attore è associata una mailbox, cioè un contenitore di messaggi ricevuti dagli altri attori ed inizialmente nel momento in cui un attore viene istanziato essa è vuota. Durante il suo ciclo di vita un attore esegue in maniera ciclica il processo di ascolto di nuovi messaggi nella mailbox ed eventuale esecuzione dell'azione corrispondente ad un messaggio in entrata;

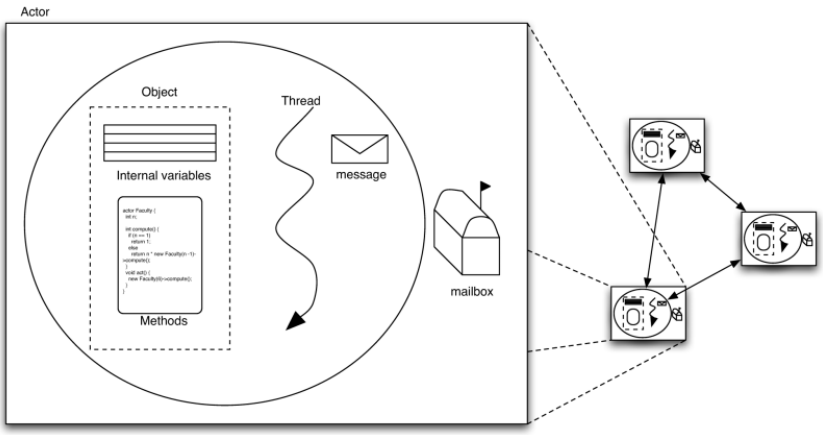


Figura 3.6: Modello ad Attori

al termine di ogni esecuzione di un azione l'attore si blocca in attesa di un nuovo messaggio in entrata e si porta in stato *idle*.

Le azioni corrispondenti ai messaggi in entrata compongono l'aspetto comportamentale dell'attore, il quale può comprendere anche un aggiornamento dello stato locale dell'attore, la creazione di nuovi attori e la spedizione di nuovi messaggi.

### Vantaggi e Svantaggi

Rispetto ad altri modelli di realizzazione di sistemi concorrenti, il modello ad attori comporta i seguenti vantaggi:

- sicurezza nell'accesso allo stato di un attore;
- non-determinismo ridotto e di conseguenza il modello è più facilmente analizzabile;
- assenza di stati e risorse condivise;
- scalabilità migliorata.

Tale modello è di conseguenza in contrasto quasi globale con il modello a memoria condivisa utilizzato dai thread, il quale:

- non presenta un concetto di interfaccia per i thread;

- presenta un grado di non-determinismo massimo che viene affrontato mediante l'utilizzo dei lock;
- presenta stati e risorse condivise;
- presenta un consumo di risorse aggiuntivo per l'utilizzo dei lock;
- presenta una gestione della scalabilità non performante.

Il modello ad attori si trova in difficoltà però con la gestione di alcuni casi specifici di sincronizzazione tra componenti di un sistema (attori), in quanto la gestione asincrona dei messaggi in arrivo attraverso la mailbox non permette nessun tipo di blocco dopo aver effettuato un'elaborazione corrispondente ad una particolare azione. Eventuali casi specifici di sincronismo sono realizzabili sfruttando il modello ad attori se esso viene supportato da meccanismi di bufferizzazione.

### 3.2.2 Generic Workers

Si parla di *Generic Workers* [7] quando si affronta in modo astratto il problema della scelta ottimale dell'architettura riguardante gli aspetti di concorrenza di un'applicazione web client. I Generic Workers possono essere utilizzati per:

- realizzare programmazione *concorrente* per sfruttare tutti i core presenti nella macchina all'interno della quale viene eseguita l'applicazione client;
- realizzare programmazione *distribuita* per eseguire più task anche su server remoti.

Questo *modello di programmazione* prevede che la comunicazione tra generic workers sia modellata con scambio di messaggi (postMessage, getMessage, ecc.), ma che sia realizzata in modo differente tra i due tipi di programmazione (da un lato possiamo immaginare una comunicazione tra thread del processo browser e dall'altro lato possiamo immaginare una comunicazione attraverso protocollo HTTP).

L'approccio ottimale consiste nel realizzare la business logic dei web worker a prescindere dal tipo di programmazione che si vuole realizzare (concorrente o distribuita) ed in seguito effettuare un wrapping con la creazione

del web worker questa volta invece dipendente dal tipo di programmazione. Un esempio applicativo banale consiste nel creare lato client due chiamate: una ad un web worker istanziato nel browser in locale ed una ad un web worker istanziato su un server remoto: tali chiamate saranno composte da una prima fase di creazione del web worker, nella quale passiamo lo stesso handler di gestione delle risposte dal server per tutte e due le chiamate ed da un ultima fase di invio dello stesso messaggio verso il web worker con la *post*; la stessa implementazione del web worker si troverà sia in locale che sul server remoto e conterrà la gestione dei messaggi ricevuti attraverso una *get*, all'interno della quale elaborerà i messaggi e risponderà con la *post*, ritornando poi in loop fino all'attesa di un nuovo messaggio o della distruzione da parte del creatore del web worker. La scelta dell'architettura dei web worker va effettuata basandosi sui fattori in gioco per questo tipo di problematiche, cioè le latenze di rete, il numero di core di una macchina, la quantità di computazione da eseguire sui web worker, ecc.

### 3.2.3 HTML5 Web Workers

Nelle nuove specifiche HTML5 è stata introdotta la possibilità di eseguire codice JavaScript in thread diversi da quello principale creato dal browser per eseguire un'applicazione web: tale funzionalità è rappresentata dai Web Workers. Si tratta di codice JavaScript inserito in un file dedicato che contiene script eseguibili con un particolare scope all'interno del quale non è consentito l'accesso agli elementi del DOM ed alcuni oggetti JavaScript (window, document e parent): questo vincolo ne permette l'utilizzo solo per ottimizzare l'esecuzione di puri script di elaborazione dati.

L'accesso all'ambiente esterno è consentito solamente attraverso scambio di messaggi con un'architettura orientata agli eventi che mette a disposizione delle API per l'invio (*postMessage*) e per la ricezione (*onMessage*) di messaggi gestiti in modo asincrono. I messaggi scambiati, a seconda dell'implementazione dei web workers presente nel browser sul quale viene eseguita l'applicazione, possono contenere stringhe o oggetti JSON serializzati.

Il sistema di message passing sul quale si basano i WebWorker per interagire tra loro, alla luce di quanto descritto precedentemente in questo capitolo, può essere ricondotto al modello ad Attori, dove i WebWorker rappresentano in questo caso gli Attori del sistema concorrente.

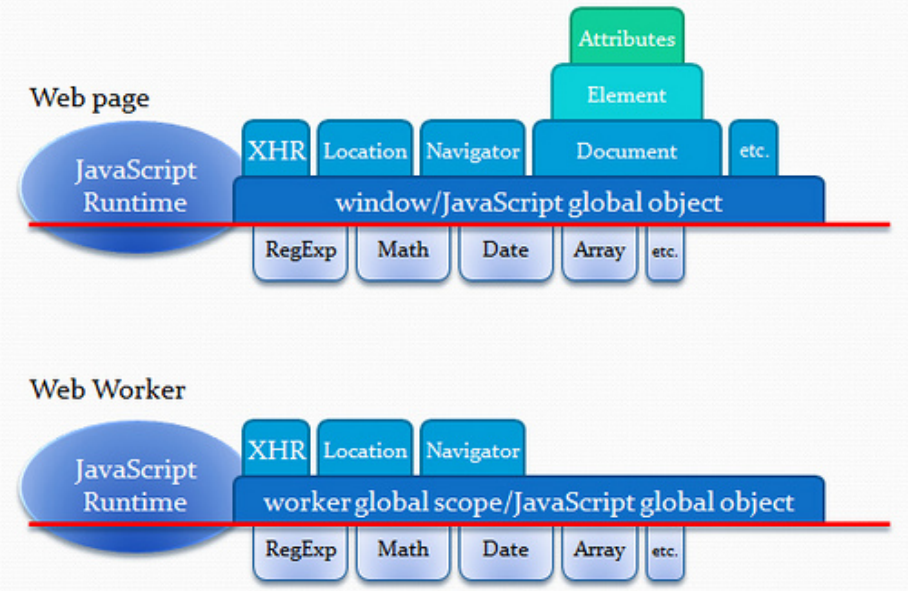


Figura 3.7: WebWorker: restrizioni.

# Capitolo 4

# Nuovi Linguaggi e Tecnologie

In questo capitolo saranno analizzate alcune tecnologie innovative nate negli ultimi anni che ruotano intorno al mondo della programmazione web. Saranno analizzati inizialmente Node.js, tecnologia che permette l'utilizzo di JavaScript lato server ed alcune librerie/Framework JavaScript ad esso correlate; successivamente sarà introdotto TypeScript un superset di JavaScript proposto da Microsoft basato su alcune novità apportate da ECMAScript 6: tale linguaggio sarà poi descritto in dettaglio nel capitolo successivo; infine sarà analizzato Dart, un nuovo linguaggio proposto da Google basato anch'esso su novità presenti in ECMAScript 6.

## 4.1 Tecnologie basate su JavaScript

Nel capitolo precedente, analizzando l'evoluzione di JavaScript, sono stati evidenziati due aspetti molto importanti, i quali stanno contribuendo al cambiamento che lo sviluppo per la piattaforma web sta avendo in questi ultimi anni: il primo consiste nella possibilità di utilizzare JavaScript come linguaggio per la programmazione lato server di un'applicazione, mentre il secondo consiste nell'utilizzare linguaggi che mediante compilazione producono codice JavaScript compatibile con i principali browser. Grazie a queste nuove tecnologie è possibile sviluppare applicazioni web utilizzando queste tipologie di linguaggi studiati appositamente per risolvere le lacune presenti su JavaScript: il processo di compilazione produce quindi codice JavaScript eseguibile sia lato client che lato server. Nei prossimi paragrafi saranno ana-

lizzate tecnologie come Node.js e TypeScript che permettono la realizzazione di tutto ciò che è appena stato descritto.

### 4.1.1 Node.js

#### Inquadramento storico

I primi anni 2000 segnarono la consacrazione del linguaggio JavaScript per la programmazione di applicazioni web client grazie all'enorme diffusione dell'utilizzo di alcune tecnologie rappresentate dal termine AJAX. Le interfacce grafiche delle applicazioni web sono evolute rapidamente e di conseguenza anche i browser con i motori JavaScript annessi al loro interno. Nel 2008 Google pubblicò sul suo browser Chrome un nuovo motore JavaScript chiamato V8 che migliorò notevolmente le performance di esecuzione di tale linguaggio superando quelle dei browser avversari. Grazie a queste evoluzioni JavaScript può ora vantare performance paragonabili ai linguaggi già esistenti utilizzabili per la programmazione server side.

#### Introduzione

Node.js [8] è una piattaforma fondata sul motore JavaScript V8 che permette la realizzazione di intere applicazioni scritte con il linguaggio JavaScript. Node.js (e anche V8) è principalmente scritto in C e C++ ed è orientato alle performance ed al basso consumo di memoria. A differenza di V8 che ha lo scopo di supportare JavaScript nei browser, Node.js ha come scopo quello di supportare processi server di lunga durata [9]. Non è la prima tecnologia che tenta di portare JavaScript sul lato server, ma soprattutto per motivi di scarse performance, tutti i tentativi effettuati in passato prima dell'avvento del motore V8 fallirono nel loro intento.

#### Modello Event-driven

JavaScript è stato concepito fin dal principio per essere un linguaggio di scripting eseguibile dai browser: tale runtime environment presenta per sua natura una gestione della sicurezza chiamata sandbox che limita le potenzialità del linguaggio per garantire alla piattaforma web un sufficiente livello di sicurezza. L'esecuzione di JavaScript su un web server non è soggetta a

questi vincoli e la piattaforma Node.js offre infatti a livello di API la possibilità di effettuare le classiche operazioni di I/O utilizzate lato server come ad esempio l'accesso ad un database ed a file; Node.js offre però a livello di linguaggio l'utilizzo di tali operazioni di I/O in modalità asincrona, seguendo il modello di riferimento event-based.

L'obiettivo di Node.js è quello di fornire un modo veloce per realizzare applicazioni web scalabili in termini di gestione delle connessioni da parte dei client verso il web server. Il sistema di gestione di tali richieste sul web server Node segue il modello *event-driven*, cioè Node viene attivato dal sistema operativo solo quando arrivano delle richieste e a quel punto alloca una piccola quantità di memoria heap ed esegue la procedura di gestione associata a tale richiesta (callback). In questo modo la gestione di tale concorrenza non deve essere effettuata con la programmazione multi-thread, la quale presenta sia un certo grado di difficoltà di realizzazione (gestione deadlock, dati condivisi tra thread, starvation, ecc.) che una peggiore gestione delle performance (2Mb di memoria nello stack allocata per ogni thread con conseguenti dispendiosi context-switch - C10K problem). Un altro aspetto importante legato al modelli di gestione asincrona delle richieste è che durante tali gestioni il processo principale non blocca mai la propria esecuzione, grazie al fatto che le operazioni di I/O come è stato anticipato in precedenza sono eseguite in modalità asincrona. L'unica criticità è rappresentata dall'eventuale presenza di operazioni di calcolo pesanti (CPU intensive), le quali potrebbero potenzialmente bloccare l'intero processo: in questo caso la scelta di un modello multi-thread rappresenta la scelta migliore in termini di reattività di risposta da parte del web server.

Per sfruttare la concorrenza nei sistemi multi-core Node mette a disposizione la possibilità di eseguire processi in parallelo utilizzando lo strumento *process*; inoltre con il *cluster module* è possibile gestire al meglio il bilanciamento delle connessioni quando si lavora con processi multipli.

### Obiettivi progettuali

Node.js è stato progettato sulla base dei seguenti obiettivi:

- gestione dell'I/O realizzata in modo asincrono, cioè non bloccante;
- streaming di dati senza l'utilizzo di meccanismi di bufferizzazione;

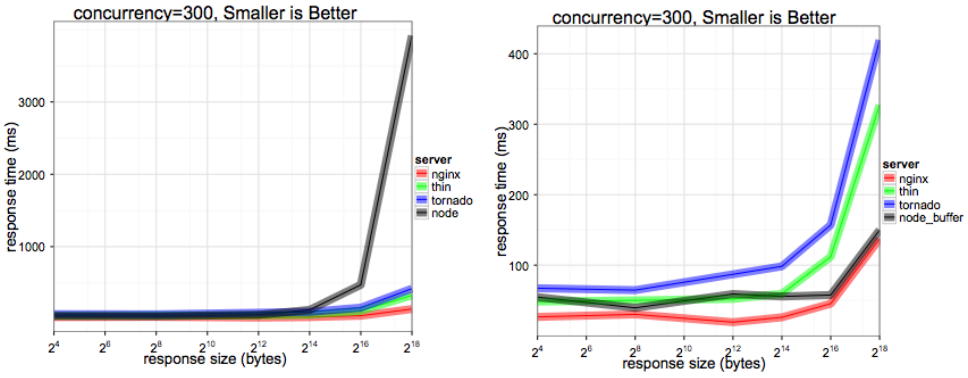


Figura 4.1: Node.js performance: utilizzo del buffer object.

- supporto ai principali protocolli TCP, DNS, HTTP: per quest'ultimo devono essere supportate anche la maggior parte delle funzionalità che esso mette a disposizione;

- il linguaggio utilizzabile per la realizzazione di applicazioni platform independent.

### Performance

Una particolarità presente nel motore V8 utilizzato da Node.js riguarda le performance legate alla gestione della memoria da parte del garbage collector: essendo quest'ultimo di tipo generazionale, colloca gli oggetti in memoria in modo casuale; di conseguenza Node, non potendo ottenere il puntatore ad un preciso dato, in fronte ad un operazione di output (ad esempio trasferimento dati verso una socket), impiega tempistiche soggette a tempi di risposta che crescono in base alla dimensione dei dati da trasferire e tali tempi paragonati a tecnologie simili presentano performance nettamente peggiori. La soluzione a tale problema consiste nell'utilizzare l'oggetto Buffer, ma nel caso di trasferimenti di grandi dimensioni il problema persiste: in figura 3.1 è illustrato un confronto con altre tecnologie concorrenti che mostra quanto è appena stato descritto.

### Librerie e Frameworks

Esiste un mondo attorno a JS composto da librerie che ne estendono le funzionalità. Stesso discorso vale per Node.js, in quanto è attiva una comunità di sviluppo che ha realizzato in questi anni molte librerie per realizzare particolari tipi di supporto (database, Network, ...) e Node.js offre un sistema di installazione di questi moduli che si occupa anche di eventuali dipendenze: *npm*. Nei prossimi paragrafi saranno analizzati alcuni progetti legati a Node.js che hanno riscosso maggior interesse.

### Realizzare un Web Server in Node.js

Un'applicazione Node.js per essere eseguita necessita l'istallazione della piattaforma Node sulla macchina ospite. È quindi necessario e sufficiente scaricare il pacchetto di installazione direttamente dal sito http://nodejs.org.

Per eseguire un programma Node.js è necessario utilizzare la shell messa a disposizione da Node stesso ed il comando che avvia l'esecuzione del programma è il seguente:

```
$ node app.js
```

Il codice corrispondente al programma stesso andrà inserito nel file con estensione .js passato come primo parametro al comando `node`.

Un semplice Web Server in ascolto di richieste http sulla porta 8080 può essere realizzato con le seguenti istruzioni:

```
var http = require('http');
var server = http.createServer(function(req, res) {
    res.writeHead(200);
    res.end('Hello World');
});
server.listen(8080);
```

La creazione del Web Server avviene utilizzando il metodo `createServer` esposto dal modulo http incluso all'inizio del programma (vedremo successivamente come avviene la gestione dei moduli). Come unico parametro viene passata una funzione anonima (callback) utilizzata dal processo principale del programma (chiamato *event-loop*) ogni qualvolta avvenga una richiesta http: tale funzione accetta a sua volta due parametri di input:

- *req*: rappresenta sotto forma di oggetto di tipo `http.ServerRequest` la richiesta http vera e propria contenente eventualmente parametri inviati verso il server;
- *res*: rappresenta sotto forma di oggetto di tipo `http.ServerResponse` il contenuto in risposta alla richiesta da inoltrare al chiamante.

### Moduli

In Node.js c'è una corrispondenza 1-1 tra file e moduli. Ogni file con estensione .js che creiamo corrisponde ad un modulo e da un altro file .js possiamo avere un riferimento a tale modulo semplicemente usando la seguente istruzione:

```
var modulo = require("modulo");
```

Come primo parametro dell'istruzione require è necessario inserire il path del modulo da utilizzare. All'interno di un modulo ogni proprietà (oggetti, metodi) che si vuole esporre deve essere inserita dentro l'oggetto `exports`. I moduli core di Node.js sono situati all'interno della directory lib. Organizzare i moduli in directory è sempre la soluzione migliore: è necessario creare una directory che conterrà i sorgenti del modulo, all'interno della quale sarà presente anche un file package.json all'interno del quale bisogna specificare il punto d'accesso del modulo (un file) e il nome ad esso associato, il tutto utilizzando il formalismo JSON; ad esempio:

```
{    name: 'modulo',
     main: './lib/module_main.js'    }
```

Organizzare il codice sorgente in moduli è una buona prassi che permette innanzitutto di velocizzare i processi di mantenimento del codice, ma soprattutto consente il riutilizzo di parti di un applicazione ove siano presenti parti comuni (ad esempio funzioni di utilità).

### Debug

Durante lo sviluppo di un'applicazione può essere necessario localizzare un problema attraverso il processo di debugging e le due modalità per poterlo eseguire sono la stampa su console di informazioni utili a tale fine o l'utilizzo

dei breakpoint. Mentre la prima modalità è realizzabile attraverso l'utilizzo dell'oggetto globale `console`, la seconda modalità è disponibile su Node.js grazie allo strumento di debugging presente sul motore V8 di JavaScript.

Se si utilizza l'IDE Eclipse per sviluppare l'applicazione Node, è disponibile un plugin per il debugger di V8; dopo averlo installato è necessario creare una configurazione per l'esecuzione del debugger, settando la porta sulla quale l'applicazione Node lavora (creando più configurazioni diverse è possibile debuggare contemporaneamente diverse applicazioni che lavorano ognuna su una porta diversa); per avviare il processo di debugging è necessario quindi avviare prima il debugger e successivamente l'applicazione, entrambi in debug mode. La particolarità di questo plugin è che l'accesso al codice sorgente da analizzare in modalità debug è fornito in copia nel progetto eclipse volatile creato al momento dell'avvio dell'applicazione Node in modalità debug: tale progetto ha come nomenclatura quella impostata nella fase di configurazione del debugger V8.

Un debugger alternativo è rappresentato dal modulo node-inspector, il quale dopo essere installato a livello globale nel sistema sul quale esso verrà eseguito, nel momento in cui viene eseguito permetterà il debugging attraverso l'utilizzo di un qualsiasi browser web accedendo all'indirizzo localhost sul quale esso è eseguito. Ovviamente l'applicazione node dovrà essere eseguita in modalità debug utilizzando lo strumento di debugging presente sul motore V8.

### 4.1.2 Express

In aggiunta alle API fornite da Node.js, per realizzare un'applicazione web vengono in aiuto allo sviluppatore un numero sempre maggiore di framework e librerie, strumenti che oltre ad aumentare le potenzialità di un applicazione, ne agevolano il processo di sviluppo.

Express è un framework basato su Node che offre un insieme robusto di utilità per realizzare agilmente applicazioni web single-page, multi-page e ibride. Molti framework (Derby, Meteor, Flatiron, ToweJS, SocketStream, ecc.) si appoggiano su Express per fornire funzionalità più specifiche (MVC, socket.io, auth, real-time, ...) rendendosi però di conseguenza spesso incompatibili con altri framework specializzati in altre funzionalità.

È un framework maturo e tra le funzionalità offerte sono inclusi: il livello middleware *connect*, il routing, la possibilità di gestire le configurazioni

dell'applicazione, un motore di templating, e molte altre funzionalità.

### Creazione di un applicazione

Vedremo ora come agevola il processo di creazione di un applicazione il framework Express.

Per prima cosa è necessario creare una directory con il nome dell'applicazione. Al suo interno deve essere creato il file `package.json` contenente le specifiche dell'applicazione tra le quali sono presenti le sue dipendenze da eventuali moduli: per utilizzare express inserire la dipendenza `''express'': versione`, dove la versione è ottenibile utilizzando il comando `npm info express version` da shell.

Sempre all'interno della stessa directory è necessario eseguire da shell il comando npm install per avviare il processo di installazione delle dipendenze (in questo caso express). Al termine della procedura eseguendo il comando `npm ls` saranno mostrate le dipendenze dell'applicazione.

Express può essere installato anche a livello di sistema per sfruttare alcune sue funzionalità tra le quali ad esempio la creazione di un applicazione. Per installare Express a livello di sistema è necessario eseguire il comando `npm install -g express`. Ad installazione avvenuta, per creare una applicazione express è disponibile il comando `express [nome app]` nel quale si possono specificare anche le dipendenze da eventuali librerie/frameworks (es: express –sessions –css stylus –ejs [nome app]). Express creerà una struttura standard per le directory dell'applicazione e creerà ogni file di configurazione necessario alle librerie specificate durante la creazione, in modo tale che basterà eseguire in successione i comandi `cd [nome app]` e `npm install` per installare ogni libreria dipendente. All'interno della directory root dell'applicazione per avviarla bastarà eseguire il comando `node app` (perché il punto di accesso del web server ris.

### Routing

Nella creazione di un'applicazione web, la gestione del routing lato server è di fondamentale importanza. Il flusso delle richieste di pagine server web in Express è il seguente:

1. generazione della richiesta della pagina web;

2. individuazione ed esecuzione del Route Handler che si occupa della gestione richiesta;

3. esecuzione del template di renderizzazione della pagina web;

4. inoltro del rendering HTML verso il client.

Di conseguenza la gestione di una richiesta di una pagina web comporta i seguenti passaggi:

1. creazione del modulo corrispondente all'handler di gestione della richiesta della pagina web all'interno della directory routes;

2. settings del routing di tale pagina inserendo nello corpo del web server le istruzioni:

   ```
   var pagina = require('./routes/pagina');
   app.get('/pagina',pagina.func);
   ```

   supponendo che il modulo corrispondente alla gestione del routing della pagina sia inserito nel file pagina.js (dentro la cartella routes) e che esporti una funzione chiamata `func`;

3. nel file pagina.js è necessario definire la callback di gestione della richiesta della pagina, all'interno della quale è possibile definire il template di rendering della pagina con il motore di templating che è stato scelto di utilizzare; ad esempio con *Jade* (nelle configurazioni dell'applicazione bisogna inserire l'istruzione `app.set('view angine', 'jade')`) bisogna creare un file `nomefile.jade` con il contenuto della pagina da renderizzare.

### Sessioni

Express è basato sul middleware connect e fornisce il supporto all'utilizzo delle sessioni. Per abilitarlo è necessario configurare l'applicazione con le seguenti istruzioni:

```
// instruzione necessaria per codificare i cookies del browser
app.use(express.cookieParser('app_cooki_key'));
app.use(express.session());
```

A questo punto nell'oggetto `request` passato come parametro alle callback delle chiamate verso il server (server pages, servizi rest, ecc.) è disponibile la proprietà `session` (ad esempio req.session) nella quale possono essere settati tutti i parametri di sessione necessari all'applicazione.

### 4.1.3 Backbone

Backbone [10] è una libreria JavaScript nata per supportare lo sviluppo di applicazioni web attraverso l'utilizzo di strumenti base che agevolano la realizzazione in modo solido e strutturato sia della parte client che della parte server.

Per la sua architettura, Backbone rientra nella categoria delle librerie MV*, in quanto implementa Model e View, ma non ha un componente Controller tradizionale, delegandone i compiti alle View e ad un componente di routing. Questo approccio è abbastanza diffuso in ambito JavaScript, dove la diversa e più complessa gestione dell'interazione utente e dello stato dell'applicazione non si adattano bene ai compiti di un controller.

I componenti base di Backbone sono:

- *Backbone.Model*: rappresenta il concetto di modello;
- *Backbone.Collection*: rappresenta il concetto di collezione di istanze di modelli;
- *Backbone.View*: rappresenta il concetto di presenter;
- *Backbone.Router*: consente la realizzazione del routing e la gestione centralizzata dello stato dell'applicazione;

È inoltre disponibile il componente Backbone.Events utile per modellare gli eventi presenti nell'applicazione e un sistema di templating per generare il contenuto HTML delle viste.

Per quanto riguarda l'installazione della libreria è sufficiente includere i sorgenti all'interno delle pagine web: Backbone è dipendente solamente dalla libreria *Underscore* la quale offre API di utilità e il motore di templating utilizzato dalle Backbone.View per renderizzare le viste dell'applicazione; anche i sorgenti di Underscore.js sono quindi da includere all'interno delle pagine.

Di seguito saranno analizzati i principali componenti della libreria Backbone mostrandone le istruzioni per l'utilizzo.

### Backbone.Model

Il Model di Backbone rappresenta un oggetto discreto contenente una serie di dati sotto forma di attributi.

In una comune applicazione non strutturata, le interazioni dell'utente (ad esempio la digitazione e l'invio dei dati) solitamente coinvolgono direttamente l'interfaccia e quindi, solo quando necessario, chiamano in causa un qualche layer di validazione e salvataggio dei dati. In un'applicazione Backbone, invece, l'interazione modifica prima lo stato di un model scatenando quindi la reazione della view che si aggiorna di conseguenza.

Per creare un modello è necessario utilizzare l'istruzione:

```
var Model = Backbone.Model.extend({...});
```

dove come parametro della funzione extend è necessario passare un oggetto contenente le configurazioni del modello.

L'istanziazione di un oggetto di un modello avviene utilizzando la keyword new:

```
var m = new Model();
```

Tra le principali configurazioni utilizzabili per la creazione di un modello troviamo:

- *initialize*: funzione eseguita all'atto della creazione di un istanza;
- *validate*: funzione che si occupa della validazione degli attributi; quando si esegue la set, se tale funzione (dopo il checks) non ritorna niente, la set va a buon fine, altrimenti ciò che ritorna è un custom error che può essere gestito nel listener per l'evento 'error' (la sua callback accetta come parametri (model,error));
- *defaults*: configurazioni di default valide per tutte le istanze del modello.

Per ogni attributo del modello sono disponibili metodi `get/set` per l'accesso a tali valori in lettura/scrittura (per accesso diretto è possibile accedere anche alla proprietà attributes direttamente sull'istanza). Il metodo `toJSON` permette di ottenere un oggetto json contenente i valori degli

attributi di un istanza. Nella funzione initialize è possibile definire listeners su eventi scaturiti sul modello attraverso la funzione on (ad esempio `this.on('evento',callback)`) ed è possibile definire anche l'attributo sul quale si vuole gestire l'evento.

Una delle caratteristiche importanti dei model di Backbone, è il fatto che vengano automaticamente dotati di un'interfaccia ad eventi emessi ad ogni modifica dei dati iniziali.

### Backbone.Collection

Una collection è un oggetto contenente una raccolta di modelli dello stesso tipo, attraverso il quale è possibile, ordinare, filtrare e manipolare i modelli contenuti.

Come per Backbone.Model, il costruttore può essere esteso con metodi e proprietà personalizzati. Solitamente basta indicare il model di riferimento e l'URL con la quale Backbone comunicherà per le operazioni di CRUD (Create, Read, Update, Delete) con il server.

Le collection offrono metodi per interagire con esse e tra i principali troviamo:

- *get*: per ottenere un istanza del modello contenuto all'interno della collection, passando come paramentro l'id (parametro che dovrà esistere per il modello);
- *add/remove*: per popolare la collection aggiungendo e rimuovendo istanze di modello.
- *on*: per definire listeners per gestire eventi scaturiti sulla collection.
- *fetch*: per recuperare dal server una collection di istanze di modello sovrascrivendo il contenuto della collection con gli elementi recuperati.
- *sync*: per sincronizzare la collection con i dati presenti sul server.
- *reset*: per eliminare il contenuto della collection inserendo nuovi modelli;

Inoltre grazie alla dipendenza da Underscore sono disponibili metodi sulle collection per gestirle al meglio: forEach, sortBy, chain (permette di agire in cascata sulla collection, ad esempio si può eseguire `filter(...).map(...).values()` per filtrare, mappare e ottenere i valori di una certa collection), ecc.

### Backbone.View

Le view in Backbone non contengono markup HTML, ma fungono da tramite fra l'interfaccia ed i modelli, definendone la logica di interazione. La parte di templating vero e proprio è demandata ad un sistema esterno.

Nel caso di Backbone, per template abbastanza semplici è possibile utilizzare il metodo `template()` fornito da Underscore. Questo metodo ricorda molto la sintassi ERB e restituisce un template precompilato al quale passare i dati come oggetti JSON. Poiché il template engine è completamente slegato da Backbone, è possibile utilizzare senza problemi un'altra libreria

Per la creazione di una vista, come per i modelli e per le collection, è necessario utilizzare la funzione extend presente nell'oggetto Backbone.View e le principali configurazioni necessarie sono le seguenti:

- *el*: è il riferimento ad un elemento del DOM rappresentante il contenitore della vista; è possibile fare riferimento ad un elemento già presente nel dom (impostando direttamente l'attributo el con il selettore CSS rappresentante l'elemento) oppure è possibile crearne uno direttamente dalla vista. L'elemento el è configurabile attraverso gli attributi: tagName, className e id;
- *render*: è la funzione che si occupa della logica di rendering della vista;
- *events*: è l'oggetto che contiene coppie `'selettore evento': callback` per gestire eventi associati agli elementi selezionati con opportune callback.

### Backbone.Router

In Backbone è possibile realizzare applicazioni single page a partire dal componente Backbone.Router. La sintassi, anche in questo caso, non differisce dai componenti precedenti e l'unica proprietà di rilievo è l'oggetto `routes`, che accetta come chiave un percorso, anche parametrico, e come valore la funzione callback da eseguire.

Dopo aver istanziato e inizializzato un router, è necessario abilitare la gestione delle routes e della navigazione con il metodo `Backbone.history.start()` che ci permette di navigare fra le pagine dell'applicazione. Il parametro `pushStart : true`, infine, abilita HTML5 History nei browser che lo supportano.

Da notare che i due comandi vengono eseguiti dopo il DOM ready, in quanto solo in questo modo saremo sicuri che la funzionalità di navigazione sia completamente funzionante su tutti i browser.

A questo punto per navigare nell'applicazione potremo sia utilizzare i normali attributi href dei link, oppure navigare sfruttare il metodo .navigate() del router:

### 4.1.4 Socket.IO

La maggior parte dei browser più diffusi fornisce il supporto all'utilizzo dei WebSockets e Socket.IO permette di realizzare applicazioni con funzionalità *real-time* (comunicazione bidirezionale client-server) su questi browser, fornendo una libreria che semplifica il più possibile i compiti allo sviluppatore, eliminando ad esempio la gestione dei meccanismi di trasporto. Nel caso in cui in un browser non fosse presente il supporto ai WebSockets, Socket.IO realizza tali funzionalità con meccanismi long-polling. Socket.IO è una libreria scritta in JavaScript e può quindi essere utilizzata per applicazioni Node.

#### Lato server

Per utilizzare Socket.IO lato server è possibile sfruttare il package manager fornito da Node.js *npm*. Sfruttando le tecnologie viste in precedenza se si utilizza il framework Express, è sufficiente inserire nel file di configurazione dell'applicazione la dipendenza da questa libreria e con il comando `npm install` eseguito dalla directory root dell'applicazione, saranno installate tutte le librerie configurate; in caso contrario sempre utilizzando npm dalla directory root dell'applicazione, eseguendo il comando `npm install socket.io` sarà installata nell'applicazione Node la libreria in oggetto.

All'interno del codice del Web Server dell'applicazione creata, per utilizzare Socket.io è necessario includere il modulo corrispondente alla libreria utilizzando il metodo require come segue:

```
var io = require('socket.io');
```

Ottenuto il riferimento al modulo, lato server questa libreria può essere utilizzata per impostare handler di gestione delle comunicazioni ricevute dai client. Per prima cosa però bisogna sempre settare la porta sulla quale

si resta in ascolto di comunicazioni con WebSocket. Dopodiché le comunicazioni da gestire saranno gestite attraverso connessioni aperte tra server e clients, quindi prima di creare gli handler delle comunicazioni è necessario creare l'handler della gestione delle connessioni. Il codice seguente mostra un esempio di gestione di comunicazioni attraverso WebSocket con Socket.io:

```
//creazione WebServer con Node o Express
//...

//WebSocket in ascolto sulla porta 80
io = io.listen(80);

// Gestione delle connessioni con i client
io.sockets.on('connection', function(socket) {

    // Gestione di una comunicazione con il client connesso
    socket.on('evento', function(dati) {

        // Inoltro della risposta verso il client connesso
        socket.emit('risposta', {dati: "..."});
    });
});
```

L'esempio mostrato sopra rende l'idea di ciò che è possibile realizzare con Socket.io: in questo è stato mostrata una semplice gestione di comunicazione client-server con risposta al chiamante; se si volesse inoltrare un messaggio broadcast verso tutti i client connessi, basterà utilizzare l'API `io.sockets.emit('message', msg)` al posto della emit utilizzata sull'oggetto socket corrispondente alla connessione aperta con il singolo client.

### Lato client

Per realizzare la controparte client di questo tipo di comunicazione con la libreria Socket.io, è necessario importare nelle pagine web in questione i sorgenti della libreria.

Per realizzare la comunicazione attraverso WebSocket, è necessario anche in questo caso aprire inizialmente la connessione con il server, in quanto l'av-

vio della comunicazione è effettuato dal client; l'apertura della connessione si effettua con la seguente istruzione:

```
var socket = io.connect("http://<uri:port>/");
```

Come è facile intuire è sufficiente passare come parametro della connect l'indirizzo URL sul quale risiede il Web Server e la porta sulla quale esso resta in ascolto delle comunicazioni mediante WebSocket.

La realizzazione della comunicazione vera e propria avviene in maniera molto simile alla controparte lato server e di seguito è mostrato un semplice esempio:

```
socket.on("connect", function() {
    // azioni da eseguire all'atto dell'apertura della connessione
});

socket.on("evento", function(dati) {

    // Inoltro della risposta verso il server
    socket.emit("risposta", {dati: "..."});
});
```

### 4.1.5 TypeScript

TypeScript [12], lanciato da Microsoft nell'ottobre del 2012 sotto licenza Open Source Apache 2.0, è un linguaggio di scripting superset di JavaScript che ne aggiunge la possibilità di utilizzare la *tipizzazione statica* ed offre il supporto per la *programmazione object oriented* attraverso l'aggiunta dei concetti di classi, interfacce e moduli. Il linguaggio è stato creato da Anders Hejlsberg, l'ideatore di C#, Delphi e Turbo Pascal. È disponibile per qualsiasi sistema operativo grazie alla possibilità di installarlo da npm.

A prima vista appare come risposta di Microsoft ad altri linguaggi che stanno nascendo in questi ultimi anni atti a risolvere i problemi presenti su JavaScript, come CoffeeScript e Dart, ma in realtà TypeScript è qualcosa di diverso: mentre i linguaggi citati precedentemente propongono una loro nuova sintassi e semantica, TypeScript estendendo sintassi e semantica di JavaScript rimane pienamente compatibile con qualsiasi script scritto con quest'ultimo linguaggio. La compatibilità con JavaScript rende TypeScript

un linguaggio più facile da apprendere rispetto ai suoi concorrenti per tutti quegli sviluppatori che hanno acquisito negli anni esperienza con JavaScript e inoltre lo rende compatibile con tutte le librerie e i framework esistenti per quest'ultimo linguaggio: questa scelta effettuata da Microsoft ha lo scopo di avvicinare a TypeScript nel minor tempo possibile il maggior numero di sviluppatori in modo tale da diffondere il proprio linguaggio, il quale sarà utilizzabile anche per sviluppare applicazioni per le proprie piattaforme desktop e mobile.

Tutte quante queste tecnologie offrono comunque la possibilità di compilare gli script in linguaggio JavaScript puro, eseguibile quindi su un qualsiasi interprete standard: la compilazione in TypeScript produce un codice finale pulito, leggibile e manutenibile non effettuando trasformazioni sui nomi delle variabili o dei metodi, né applicando una minimizzazione dell'output.

Ciò che di TypeScript è stato aggiunto a JavaScript è parte fondamentale dello scopo per il quale questo linguaggio è nato, cioè offrire strumenti che agevolino lo sviluppatore nel costruire applicazioni complesse e scalabili: il tutto può essere ottenuto con i costrutti offerti da TypeScript direttamente, ma anche con tool a supporto del linguaggio come quelli esistenti per altri linguaggi object oriented che non possono esistere per JavaScript a causa della sua natura. Esistono plugin per diversi ambienti, quali SublimeText, VI ed Emacs, ma è in Visual Studio 2012 che TypeScript trova il suo habitat naturale.

## 4.2 Nuove Tecnologie

Nei precedenti paragrafi sono state analizzate tecnologie che si appoggiano direttamente sul linguaggio JavaScript per fornire il proprio supporto allo sviluppo di applicazioni web; di seguito sarà analizzato Dart, un nuovo linguaggio che si differenzia dalle precedenti tecnologie per avere una sintassi e semantica propria, ma che offre comunque il supporto alla compilazione in JavaScript per risolvere momentaneamente il problema di compatibilità mancante con i principali browser web.

### 4.2.1 Dart

Dart [11], lanciato da Google nell'ottobre del 2011 (un anno prima di TypeScript), è un linguaggio class-based nato con l'obiettivo di rimpiazzare

JavaScript come linguaggio per la programmazione web, ritenendosi più idoneo soprattutto per la realizzazione di applicazioni strutturate. L'intento di Google era quello di realizzare un linguaggio che riuscisse a risolvere i problemi che JavaScript, anche attraverso ulteriori evoluzioni, non sarebbe stato in grado comunque di risolvere, il tutto offrendo migliori performance, tool per agevolare la programmazione e un supporto maggiore per gli aspetti legati alla sicurezza.

Dart è quindi un linguaggio che permette la realizzazione di sistemi web strutturati, flessibili, efficienti e scalabili, il tutto attraverso costrutti naturali e facili da apprendere ed accompagnato da tools in grado di rendere più agile lo sviluppo; inoltre per essere al passo con le nuove tecnologie, Dart è stato concepito per essere un linguaggio appropriato per tutto il range di dispositivi attualmente in commercio (telefoni, tablet, computer, ecc.).

L'obiettivo principale di Dart, cioè supportare la programmazione web strutturata, è raggiunto grazie alle sue caratteristiche di linguaggio object oriented class-based con il supporto alla tipizzazione statica, proprio come TypeScript, con la differenza che Dart, essendo basato su una propria sintassi e semantica, si ritrova a dover affrontare il passo iniziale di diffusione del linguaggio, su un mercato ricco di alternative; Google ha optato per due soluzioni atte a risolvere al meglio questo problema:

1. *Performance migliori*: Dart è stato realizzato puramente come nuovo linguaggio, senza nessun tipo di dipendenza da altre tecnologie e Google ha realizzato un motore di esecuzione per tale linguaggio che per ora è disponibile solo per il browser Chromium (browser dal quale deriva Chrome); il tutto è stato realizzato con l'obiettivo di ottenere prestazioni migliori di quanto il mercato dello sviluppo web possa offrire al momento.

2. *Compilazione in JavaScript*: non essendo disponibile il motore per l'esecuzione di Dart al momento per i principali browser, Google ha realizzato un compilatore in grado di produrre sorgenti scritti in JavaScript, in modo tale che un'applicazione scritta in Dart possa essere eseguita comunque sui principali browser.

L'obiettivo principale di Dart è quindi di riuscire a conquistare il mercato dello sviluppo web con performance migliori rispetto a JavaScript in aggiunta a delle caratteristiche più avanzate, facendone il sostituto definitivo

di JavaScript con una conseguente diffusione del proprio motore esecutivo sui principali browser e sui principali dispositivi in commercio.

### Isolate

Dart fornisce il supporto alla programmazione concorrente attraverso il concetto di *Isolate*, basandosi sul modello ad attori (paragrafo 3.2.1), dove un Isolate corrisponde ad un Attore facente parte di un sistema concorrente; seguendo i principi di questo modello, ogni Isolate possiederà un proprio stato non condiviso con nessun altro Isolate, il quale potrà quindi essere modificato solamente attraverso scambio di messaggi.

Attraverso l'utilizzo degli Isolate è possibile quindi realizzare applicazioni che godono dei vantaggi offerti dalla programmazione concorrente, cioè:

- migliori performance;
- responsività;
- robustezza.

In Dart il modello di comunicazione tra Attori è esteso dando la possibilità a ciascun Isolate di avere più di una mailbox referenziata da una specifica *porta* che può essere di due tipi: uno per mailbox utilizzate per spedire messaggi e uno per riceverli; in questo modo è garantito un sufficiente livello di sicurezza che vieta ad un Isolate di poter leggere messaggi su mailbox di altri Isolate atte alla ricezione. Una porta (mailbox) per la ricezione di messaggi può essere definita in maniera implicita (quando un Isolate viene creato da un altro Isolate viene creata automaticamente una porta di ricezione messaggi) o esplicita (istanziando direttamente un oggetto di classe `ReceivePort`). Le porte per l'invio dei messaggi (`SendPort`) sono collegate ciascuna ad una specifica ReceivePort e per ottenerle è necessario utilizzare il metodo `toSendPort()` sull'oggetto che referenzia la porta di ricezione.

I messaggi ricevuti in una specifica porta sono gestiti in modo asincrono attraverso il metodo `receive()` il quale in input necessita una funzione di callback: tale funzione come signature prevede un messaggio e una SendPort del mittente sulla quale inoltrare eventuali messaggi di risposta con il metodo `send()` che presenta la stessa signature della callback illustrata precedentemente.

La creazione di un'Isolate può avvenire estendendo la classe `Isolate` oppure eseguendo il metodo `spawn()` su un oggetto di tipo Isolate: quest'ultima modalità crea un Isolate in modo asincrono (in linea con il modello ad Attori) ed a creazione avvenuta viene chiamata una funzione `future`, alla quale viene passato in input una SendPort utilizzabile dall'Isolate che ha eseguito il metodo spawn() per poter comunicare con l'Isolate creato.

Per quanto riguarda la creazione di un Isolate esistono due tipi di costruttori ognuno con un proprio nome assegnato (*named constructor*):

- *light*: un Isolate creato con costruttore light viene eseguito sullo stesso thread dell'Isolate creante;
- *heavy*: un Isolate creato con costruttore heavy viene eseguito in un thread dedicato.

Entrambi i tipi di costruttore implicano la creazione di Isolate con proprio stato non condivisibile e modificabile quindi solo attraverso scambio di messaggi, ma per quando riguarda l'esecuzione del codice appartenente ad un thread nel caso di Isolate light in un istante può essere eseguito solo un Isolate alla volta, mentre nel caso di Isolate heavy l'esecuzione può essere concorrente.

# Capitolo 5

# TypeScript: Approfondimento

In questo capitolo sarà approfondita la conoscenza del linguaggio TypeScript introdotto nella sezione 4.1.5; ne saranno analizzate le caratteristiche attraverso anche alcuni esempi di utilizzo.

## 5.1 Installazione

Il compilatore di TypeScript essendo implementato in TypeScript può essere utilizzato in qualsiasi ambiente sul quale è presente un motore JavaScript. La tecnologia Node.js attraverso il node package manager (npm) permette l'installazione del compilatore TypeScript a linea di comando come Node.js package, utilizzando il seguente comando:

```
npm install -g typescript
```

È disponibile per l'IDE Visual Studio 2012 un plugin per TypeScript che contiene il compilatore e tool che offrono una ricca esperienza di sviluppo.

## 5.2 Compilazione

Il comando disponibile per compilare sorgenti TypeScript (nell'esempio `example.ts`) è il seguente:

```
tsc example.ts
```

Se il processo di compilazione va a buon fine, il risultato genererà il file example.js contenente il corrispondente codice JavaScript puro, privo dei costrutti aggiunti da TypeScript (tipizzazione statica e OOP).

## 5.3 Tipizzazione statica opzionale

TypeScript offre la possibilità di utilizzare la tipizzazione statica, la quale può essere realizzata attraverso sintassi post-fissa, poco familiare per chi utilizza linguaggi C-like, ma adatta all'opzionalità come avviene appunto in questo caso.

L'utilizzo della tipizzazione statica opzionale comporta una serie di vantaggi:

- è possibile realizzare il checking sui tipi di dato a livello di compilazione;
- le interfacce sono realizzate con implementazione "Structured Type System", un approccio non invasivo che unito alla tipizzazione statica opzionale, permette la creazione di oggetti senza specificare la loro interfaccia, ma semplicemente rispettando la loro definizione;
- è possibile realizzare tool per offrire auto-completamento del codice;
- il processo di refactoring è facilitato.

La definizione del tipo di una variabile può essere effettuata con la seguente sintassi:

```
var word:string;
```

Variabili, proprietà e parametri di input o di output di un metodo, possono essere definiti con i tipi primitivi presenti in TypeScript o con classi/interfacce personalizzate.

Tra i tipi primitivi troviamo:

- *number*: numeri in virgola mobile a precisione doppia (64bit);
- *boolean*: valori logici booleani (true/false);
- *string*: sequenze di caratteri Unicode UTF-16;

- *null*: l'equivalente dello stesso tipo definito in JavaScript;
- *undefined*: l'equivalente dello stesso tipo definito in JavaScript.

Per i parametri di ritorno delle funzioni è disponibile anche il tipo primitivo void. Le funzioni possono essere definite utilizzando le due seguenti modalità:

```
funzione: (parametro :tipo, ...) => tipo;
funzione(parametro :tipo, ...) :tipo;
```

## 5.4 Object Oriented Programming

TypeScript offre il supporto alla programmazione object oriented class-based attraverso i costrutti `class` e `interface` grazie ai quali rispettivamente possono essere create classi e interfacce. Per quanto riguarda l'utilizzo delle classi, non mancano per JavaScript le possibilità di potere essere realizzate, ma TypeScript semplifica notevolmente questo aspetto soprattutto per quanto riguarda la possibilità di effettuare ereditarietà.

Le creazioni di interfacce e un classi in TypeScript avvengono nel seguente modo (seguendo lo standard ECMAScript 6):

```
interface Persona {
    nome: string;
    cognome: string;
    cod_fiscale?: string;
}

class Studente {
    private email : string;
    constructor(public nome, public cognome) {
        this.email = nome + "." + cognome + "@studio.unibo.it";
    }
}

var user: Persona = new Studente("Enrico", "Gramellini");
```

Il precedente codice è corretto in quanto lo Structured Type System di TypeScript fornisce il type checking basato non solo sul nome dei tipi, ma anche sulla loro struttura: ad esempio nel codice precedente viene definita una variabile di tipo Persona, la quale viene inizializzata con un oggetto di tipo Studente che possiede all'interno della propria struttura delle caratteristiche (proprietà) definite anche nell'interfaccia Persona (le variabili definite nella signature del costruttore di una classe con visibilità specificata, diventano proprietà di quella classe).

La compilazione dei costrutti OOP forniti da TypeScript produrrà codice JavaScript object oriented prototype-based, il quale non prevedendo la possibilità di distinguere classi da interfacce, non presenterà nessuna controparte JavaScript per quanto riguarda le interfacce, quindi esse saranno utilizzate esclusivamente dal compilatore per validare il codice sorgente.

È importante sottolineare che per la dichiarazione di una variabile è necessario l'utilizzo della direttiva `var`, mentre per la definizione di una proprietà di un'interfaccia o di una classe essa non deve essere utilizzata.

Per quanto riguarda le proprietà e i metodi definibili per una classe è possibile specificarne la **visibilità** attraverso le direttive `public` e `private`; la possibilità di definire una proprietà o un metodo `protected` non è disponibile a livello di linguaggio in quanto è programmata per la versione ECMAScript 7, mentre TypeScript al momento è basato sulle funzionalità presenti in ECMScript 6.

In una classe la definizione di un metodo può avvenire con o senza implementazione in modo esclusivo; attualmente non essendoci la possibilità di definire una classe astratta, nel caso un metodo venga solo definito e successivamente utilizzato su un istanza di tale classe, il compilatore non causerà errore, ma esso si verificherà a runtime se non è presente nessuna implementazione in eventuali classi dalla quale essa estende.

Le proprietà definibili per un'interfaccia possono essere definite come opzionali aggiungendo la direttiva `?` subito dopo il nome. L'opzionalità di una variabile può essere definita anche nella signature di un metodo sempre con la stessa modalità; in questo modo è possibile quindi definire l'**overloading** di un metodo con un unica implementazione: questa è anche l'unica possibilità offerta da TypeScript per realizzare questo meccanismo, in quanto utilizzando più implementazioni diverse dello stesso metodo con signature differenti comporterebbe l'errore di compilazione `duplicate identifier`; l'overloading in TypeScript è un costrutto utilizzato esclusivamente in fase

di compilazione e l'intero meccanismo di controllo dei parametri, anche se supportato comunque dal controllo statico dei tipi dal compilatore, rimane a carico dello sviluppatore.

Per definire di una classe che realizza un'interfaccia esistente è necessario utilizzare la direttiva `implements` subito dopo il nome.

### 5.4.1 Ereditarietà

L'introduzione delle classi all'interno del linguaggio comporta anche il vantaggio di poter usufruire di altre peculiarità fondamentali della programmazione orientata agli oggetti, come l'ereditarietà, la quale rende possibile definire relazioni di generalizzazione/specificazione tra due o più classi.

Per realizzare ereditarietà tra classi, all'atto della creazione di una classe, dopo il suo nome è necessario specificare da quale classe essa deve estendere proprietà e metodi utilizzando la direttiva `extends`.

```
class Animale {
    cibo: string;
    constructor(public nome: string) {
        this.cibo = "crocchette";
    }
    mangia(unità: number) {
        alert(this.nome + " mangia " + unità + " " + this.cibo);
    }
}

class Topo extends Animale {
    constructor(nome: string) {
        super(nome);
        this.cibo = "formaggini";
    }
    mangia() {
        super.mangia(5);
    }
}

class Gatto extends Animale {
```

```
    constructor(nome: string) {
        super(nome);
    }
    mangia() {
        super.mangia(10);
    }
}

var jerry = new Topo("Jerry il topo");
var tom = new Gatto("Tom il gatto");

jerry.mangia();
tom.mangia();
```

### Chaining APIs problem

TypeScript attualmente è in fase alpha (developer preview) e presenta ancora molti bug, segnalati e discussi all'interno della sezione dedicata sul sito codeplex, relativo al progetto TypeScript. Tra i bugs attualmente presenti ne esiste uno relativo alla gestione delle chiamate in cascata tra api. Prendendo come esempio di riferimento il codice sottostante è possibile verificare il comportamento anomalo del compilatore:

```
class Base {
    m1(){
        return this;
    }
}

class Derivata extends Base {
    m2(){
        return this;
    }
}

new Derivata().m1().m2();
```

La classe `Derivata` estende la classe `Base` e ne eredita ogni su metodo (in questo caso solo `m1`) correttamente: tale metodo, ritornando il riferimento

al contesto della classe stessa attraverso l'utilizzo di `this`, richiamato da un'istanza della classe Derivata dovrebbe ritornare il riferimento al contesto di tale classe; su un metodo che ritorna un riferimento ad un contesto di una classe sono visibili tutti i metodi esposti da tale classe e in questo caso dalla classe Derivata dovrebbero essere visibili entrambi i metodi `m1` e `m2`: il risultato inaspettato dal processo di compilazione consiste nel restituire un errore nell'esecuzione di m2 in catena al valore ritornato da m1, in quanto la `return this` di m1 non ritorna effettivamente il contesto della classe dalla quale è stato eseguito, ma ritorna quello della classe nella quale è definito, cioè la classe Base.

Questo è un grave errore attualmente presente in TypeScript in quanto non è rispecchiato il giusto comportamento che l'ereditarietà dell'OOP richiede, limitandone di conseguenza le potenzialità del linguaggio stesso. Nella discussione relativa a questo bug un utente ha trovato però un metodo per aggirare momentaneamente il problema aggiungendo nella classe Derivata la definizione del metodo m1 nel modo seguente:

```
...
m1: () => Derivata;
...
```

In questo modo il compilatore non genera più l'errore perché nella classe Derivata il metodo m1 è definito in modo tale che restituisca un oggetto della stessa classe.

### 5.4.2 Polimorfismo

Un'altra caratteristica importante della programmazione Object Oriented è il polimorfismo, pienamente supportato da TypeScript grazie anche alla possibilità di utilizzare le asserzioni di tipi all'interno delle espressioni; nell'esempio sottostante sarà illustrato un caso tipico di utilizzo del polimorfismo:

```
class Figura {
    ...
}

class Cerchio extends Figura {
    ...
```

```
}

function creaFigura( tipo: string ): Figura {
    if ( tipo === "cerchio" ) return new Cerchio();
    ...
}
var cerchio = <Cerchio> creaFigura( "cerchio" );
```

In questo caso in presenza di una funzione polimorfica, possiamo utilizzarla nell'assegnamento di un valore ad una variabile asserendo il tipo di ritorno della funzione in base alla business logic che vogliamo applicare in tale assegnamento.

## 5.5 Moduli

TypeScript permette la creazione di moduli, concetto utile per strutturare il codice sorgente e per sfruttare la riusabilità. Un modulo possiede un nome ed un insieme di concetti TypeScript (classi, interfacce, variabili, funzioni, ecc.); tutto ciò che si vuole rendere utilizzabile di un modulo dal componente che lo utilizza va dichiarato con la keyword `export` all'inizio della definizione del costrutto stesso. Di seguito sarà mostrato un esempio di realizzazione e utilizzo di un modulo:

```
module MathModule {
    export class Expression {
        sum(term1: number, term2: number) {
            return term1 + term2;
        }
    }
}
var expr = new MathModule.Expression();
var res = expr.sum(1,2);
```

Una delle caratteristiche più apprezzate di TypeScript riguarda proprio i moduli, in quanto esso offre la possibilità di generare il codice JavaScript corrispondente in due formati diversi, cioè *CommonJS Modules* and *Asynchronous Module Definition* (AMD): CommonJS è un pattern JavaScript utilizzato per realizzare codice modulare per ambienti server come ad esempio

Node.js e AMD è una sua estensione che consente il caricamento asincrono dei moduli, tipicamente utilizzato nei browser web. TypeScript permette di realizzare la logica applicativa astraendo dall'ambiente per il quale essa è realizzata, in quanto la scelta del tipo di generazione del codice deve essere effettuata all'atto della compilazione, utilizzando la direttiva `--module` e non è quindi necessario agire aggiungendo codice aggiuntivo cosiddetto *boilerplate*; di seguito sarà mostrato come TypeScript compilerà il codice dell'esempio precedente inserito nel file `example.ts` - composto dalla definizione di un'interfaccia, di una classe e di un'istanza di tale classe la quale sarà esportata - in seguito all'utilizzo della direttiva `--module` per impostare il formato desiderato di output:

```
interface Persona {
    nome: string;
    cognome: string;
    cod_fiscale?: string;
}

class Studente {
    private email : string;
    constructor(public nome, public cognome) {
        this.email = nome + "." + cognome + "@studio.unibo.it";
    }
}

export var user: Persona = new Studente("Enrico", "Gramellini");
```

Per ottenere codice JavaScript con formato **CommonsJS** (formato di default) eseguire il comando tsc senza direttiva `--module`, oppure:

```
tsc --module commonjs example.ts
```

Tale comando produrrà il seguente codice JavaScript:

```
var Studente = (function () {
    function Studente(nome, cognome) {
        this.nome = nome;
        this.cognome = cognome;
```

```
        this.email = nome + "." + cognome + "@studio.unibo.it";
    }
    return Studente;
})();
exports.user = new Studente("Enrico", "Gramellini");
```

Per ottenere codice JavaScript con formato **AMD** eseguire invece il comando:

```
tsc --module amd example.ts
```

Tale comando produrrà il seguente codice JavaScript:

```
define(["require", "exports"], function(require, exports) {
    var Studente = (function () {
        function Studente(nome, cognome) {
            this.nome = nome;
            this.cognome = cognome;
            this.email = nome + "." + cognome + "@studio.unibo.it";
        }
        return Studente;
    })();
    exports.user = new Studente("Enrico", "Gramellini");
})
```

È necessario fare distinzione tra moduli interni ad un file ed esterni: in una tipica applicazione che abbia un minimo livello di complessità, il codice sorgente è organizzato su più file diversi ed ogni file può essere visto come modulo se esso esporta attraverso la keyword export qualche suo componente; all'interno di un file sorgente possono essere definiti anche moduli interni attraverso il costrutto `module` visto in precedenza. Per quanto riguarda i moduli interni possono essere referenziati direttamente dal nome ad essi associato, mentre per quanto riguarda i moduli esterni, l'inclusione di un modulo all'interno di un altro avviene semplicemente eseguendo il seguente comando TypeScript in questo caso eseguito per importare dal file `example.ts` la variabile `user` esportata:

```
import example = module('example');
alert(example.user.email);
```

Il codice JavaScript generato con il formato AMD sarà compatibile con tale pattern di modularizzazione del codice, il quale però necessita l'utilizzo della libreria *RequireJS* per poter essere eseguito correttamente; la descrizione di tale libreria non rientra nel focus di questa tesi, ma nel prossimo capitolo descrivendo un caso di studio realizzato sarà mostrato come essa è stata utilizzata per gestire le dipendenze dell'applicazione client dalle librerie di terze parti da essa utilizzate.

Per quanto riguarda il codice JavaScript generato con il formato CommonsJS, esso è direttamente compatibile con la piattaforma Node.js in quanto essa implementa tale pattern.

## 5.6 Debug

Come avviene per altri progetti *compile-to-javascript*, anche TypeScript soffre la mancanza di strumenti per effettuare il debug direttamente sul codice sorgente prodotto; in questo caso il codice JavaScript prodotto è leggibile e quindi su di esso può essere effettuato il debug con gli strumenti già esistenti.

Non tutto il codice TypeScript però è sempre facilmente sottoponibile a debug utilizzando la versione JavaScript prodotta: fortunatamente esiste la tecnologia *SourceMaps* che permette di effettuare il mapping tra sorgenti di linguaggi diversi. TypeScript implementa tale funzionalità a livello di compilazione utilizzando la direttiva `-sourcemap` che permette la produzione di un file aggiuntivo con estensione `.map` per ogni file sorgente compilato; il tool che permette di effettuare il debug dell'applicazione JavaScript, se è compatibile con la tecnologia SourceMaps, permetterà in presenza di tali file con estensione .map, di effettuare il debug direttamente sui sorgenti TypeScript.

## 5.7 Utilizzo di librerie JavaScript

Una caratteristica importante di TypeScript è che esso è compatibile con tutte le librerie esistenti per JavaScript. Una libreria può essere utilizzata in un sorgente TypeScript senza nessuna aggiunta di ulteriore codice, ma deve essere necessariamente creato un file con estensione `.d.ts` contenente

la definizione dei tipi di dato e delle interfacce utilizzate da tale libreria; in alcuni casi, in presenza di librerie che forniscono utilità attraverso un oggetto globale, nel file d.ts sarà presente anche la dichiarazione di tale variabile utilizzabile all'interno dell'applicazione TypeScript per fare riferimento quindi alla libreria stessa (come ad esempio succede per la nota libreria JQuery).

Tali file devono essere inclusi in cima al file contenente il punto d'accesso dell'applicazione utilizzando il meccanismo di referenziazione di file esterni che prevede l'inserimento di un particolare commento chiamato *reference comment* definito come segue:

```
/// <reference path="path/to/library.d.ts" />
```

Ad esempio per utilizzare la libreria JQuery, sarà necessario creare il file jquery.d.ts all'interno del quale dovranno essere definiti i tipi di dato e le interfacce fornite da tale libreria e come ultima istruzione dovrà essere definita la variabile $ per poter utilizzare la libreria correttamente all'interno dei sorgenti TypeScript. Tale file dovrà essere poi referenziato all'interno del file contenente il punto d'accesso dell'applicazione.

Questo meccanismo di referenziazione delle interfacce delle librerie di terze parti utilizzate serve al compilatore solo per effettuare i controlli sui tipi di dato e quindi non produrrà nessun sorgente JavaScript corrispondente a tali file con estensione d.ts. Si noti che il compilatore TypeScript essendo sviluppato a sua volta con il linguaggio Typescript stesso, fa uso di un file chiamato lib.d.ts il quale contiene la definizione di JavaScript.

È disponibile una repository github (https://github.com/borisyankov/DefinitelyTyped) nella quale sono censite le librerie JavaScript più note e se per un'applicazione è necessario utilizzare una libreria tra quelle censite, basterà recuperare il file d.ts corrispondente ed includerlo nel proprio progetto.

Infine è importante sottolineare che il compilatore TypeScript fornisce la possibilità di generare automaticamente un file di definizione di una libreria custom aggiungendo nel comando di compilazione la direttiva `{declarations`. Ad esempio se abbiamo prodotto nel file sorgente `example.ts` una libreria che vogliamo riutilizzare in un altro progetto, compilando utilizzando tale direttiva, avremo la possibilità sia di riutilizzarla che di sfruttare le potenzialità dei tools eventualmente disponibili per il linguaggio.

## 5.8 Tools

JavaScript a causa della sua natura di linguaggio object-based con tipizzazione dinamica, soffre la mancanza di tool che possano supportare lo sviluppatore nella scrittura del codice. TypeScript essendo invece un linguaggio class-based con tipizzazione statica opzionale è compatibile con i tipici tool presenti per i linguaggio dello stessa tipologia più diffusi: tra i tool più comuni troviamo:

- sottolineatura della sintassi;
- auto-completamento del codice;
- rilevamento di errori;
- intellisense;
- ...

Attualmente tali tools sono disponibili solo per un numero limitato di IDE, cioè Visual Studio 2012 (attraverso un apposito plugin), Vi, Emacs e Sublime Text. Sul sito ufficiale di TypeScript è presente in aggiunta una sezione atta a supportare il training del linguaggio direttamente dal browser attraverso un editor di testo: tale editor è fornito dei tool descritti in precedenza e questo significa che essi sono realizzabili anche con il linguaggio JavaScript stesso in quanto sono eseguiti su browser.

# Capitolo 6

# Caso di Studio Applicativo

In questo ultimo capitolo sarà mostrato come cambia lo sviluppo di un sistema software Web utilizzando il linguaggio TypeScript per supportare la strutturazione del codice sorgente altrimenti scritto interamente in JavaScript. L'enfasi di questa tesi è posto principalmente sulla fase del ciclo di vita del software riguardante lo sviluppo, quindi saranno tralasciati dal punto di vista puramente ingegneristico gli aspetti di analisi dei requisiti e di progettazione del sistema, mentre saranno analizzati tutti gli aspetti riguardanti le tecnologie utilizzate durante lo sviluppo, dal punto di vista dell'utilizzo di TypeScript al posto di JavaScript puro.

## 6.1 UpTennis

Per analizzare gli aspetti legati al sistema software anche solo dal punto di vista i TypeScript, è necessario introdurlo per conoscere i concetti fondamentali che ruotano attorno ad esso.

UpTennis (http://www.uptennis.com) è un Social Network che permette a sportivi legati al mondo del Tennis e del Beach Tennis di riunirsi giocando partite valide ai fini di una classifica stilata stagionalmente: tale classificazione permette al sistema di individuare a fine stagione (la quale ha la durata di un anno solare) i migliori giocatori che potranno partecipare ad un torneo finale. I giocatori si potranno registrare al sistema inserendo dati personali che li identificheranno all'interno di esso e potranno iscriversi ad una classifica in base allo sport che praticano ed al livello su di esso conseguito. Per

ogni partita giocata, in base al risultato finale, saranno attribuiti punti che contribuiranno a definire appunto le classifiche.

Il sistema sarà utilizzabile attraverso la piattaforma Web e permetterà la navigazione di alcune viste sia per utenti registrati che per semplici visitatori. Tra gli utenti registrati al sistema si possono distinguere utenti amministratori del sistema, utenti utilizzatori e utenti gestori di circoli sportivi; questi ultimi utenti avranno la possibilità di accedere al sistema per presentare il proprio circolo sportivo utilizzabile dai giocatori per organizzare partite ed eventualmente per comunicare con questi ultimi. Il sistema dovrà presentare un'interfaccia grafica che permetterà agli utenti di effettuare tutte le operazioni fornite da esso.

Il modello di rilascio del sistema in produzione dovrà seguire un particolare flusso per motivi legati alle necessità nelle tempistiche di utilizzo del sistema stesso: nella prima release dovranno essere presenti funzionalità base, alle quali saranno aggiunte con le successive release tutte le funzionalità commissionate ed eventuali nuove emerse con l'utilizzo del sistema da parte degli utenti. Di conseguenza la progettazione del sistema dovrà tenere in considerazione questi fattori per ottimizzare al meglio lo sviluppo del software.

In questa tesi saranno analizzate le problematiche affrontate durante lo sviluppo della prima release del sistema, la quale ruota fondamentalmente intorno a due macro entità: gli **utenti** del sistema e le **partite** organizzate. L'interfaccia grafica dovrà presentare le seguenti **viste**:

Viste con contenuti statici:

- Home page di benvenuto ai visitatori;
- Regolamento del sistema;
- Programma torneo Master finale;
- Privacy Policy;
- Contatti;
- Creazione partita.

Viste con contenuti dinamici:

- Profilo utente registrato: sarà possibile visualizzare di ciascun utente in modo pubblico statistiche relative alle partite effettuate, dati relativi allo sport per il quale si è iscritto, partite organizzate/effettuate e dati relativi all'utente;
- Classifiche: saranno consultabili le tre classifiche stagionali (Tennisti esperti, Tennisti amatoriali e Giocatori di Beach Tennis);
- Elenco circoli: saranno consultabili dati di contatto sui circoli sportivi aderenti all'iniziativa;
- Elenco partite: saranno elencate le partite organizzate dai giocatori e sarà consultabile l'archivio storico;
- Dettaglio partita: sarà possibile visualizzare i partecipanti alla partita, il risultato (se disponibile) e altri dati relativi alla partita stessa.

Gli utenti registrati al sistema per la prima release saranno solamente i giocatori che parteciperanno alle partite, mentre successivamente saranno introdotti anche i gestori dei circoli sportivi e gli amministratori, con l'aggiunta di ulteriori viste atte a supportare ulteriori features.

Gli utenti registrati (i giocatori) avranno a disposizione strumenti per poter eseguire le seguenti **operazioni**:

- Registrazione al sistema con inserimento dati relativi ad uno sport;
- Inserimento immagine del profilo;
- Organizzazione di una partita;
- Partecipazione/Abbandono ad/di un una partita;
- Invito di giocatori a partecipare ad una partita organizzata;
- Inserimento/Conferma risultato di una partita.

L'accesso al sistema sarà possibile eseguirlo mediante classico iter (inserimento email e conferma di attivazione attraverso link inserito in un messaggio di posta elettronica) oppure utilizzando un account Facebook, integrando in futuro anche funzionalità legate a questo social network.

Infine il sistema dovrà eseguire periodicamente le seguenti azioni in background (**batch**):

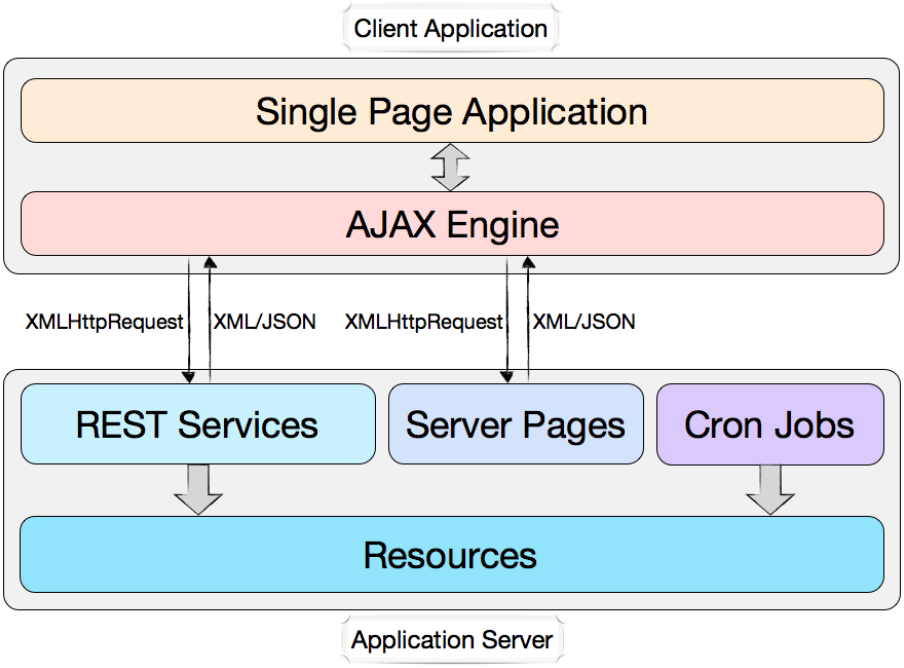


Figura 6.1: Macro-Architettura Applicazione UpTennis

- Aggiornamento delle classifiche calcolando i punteggi degli utenti in base alle partite concluse con risultato inserito.
- Controllo di presenza di partite terminate con risultati non inseriti da parte dei giocatori partecipanti e conseguente inoltro di email di avviso ai partecipanti stessi.

## 6.2 Architettura Software

Nel capitolo 2 abbiamo visto che l'architettura di riferimento per le applicazioni web segue di base il classico modello client/server e a seconda delle necessità e delle caratteristiche dell'applicazione, tale architettura sposa meglio un pattern di modellazione piuttosto che un altro.

Le caratteristiche di UpTennis richiedono principalmente la memorizzazione di dati (risorse) sia lato server che lato client, nel primo caso in modo persistente, mentre nel secondo non essendo presenti funzionalità offline, in

modo temporaneo. Su tali dati all'interno del flusso di esecuzione dell'applicazione, saranno effettuate operazioni di lettura e scrittura su funzionalità attivate sia lato client che lato server.

La parte client dovrà garantire una ricca esperienza d'utilizzo agli utenti del sistema ed uno dei pattern che meglio sposa questo tipo di applicazioni è il pattern REST, il quale segue il modello più generale SOA; tale pattern struttura l'applicazione in modo tale che la parte client e la parte server possano comunicare attraverso servizi che utilizzano le classiche POST, GET, PUT e DELETE per performare operazioni cosiddette CRUD (Create, Read, Update e Delete).

Lo strato server quindi sarà composto da servizi REST, da server pages e dalle operazioni batch descritte in precedenza che saranno eseguite a cadenza regolare di tempo; per quanto riguarda le server pages, nella prima release non è stata prevista una versione dell'applicazione multi-page per browser con JavaScript disabilitato, quindi sarà presente una singola server page corrispondente ad un documento HTML vuoto che conterrà l'applicazione single-page renderizzata poi lato client. Lo strato client invece sarà strutturato come applicazione *single page*, la quale comporta un iniziale caricamento dei sorgenti necessari all'esecuzione e una successiva fase di utilizzo che scaturisce comunicazioni con lo strato server attraverso l'utilizzo dei servizi REST descritti in precedenza: la peculiarità di questo tipo di applicazioni è l'assenza del refresh delle pagine web che compongono l'applicazione, in quanto esse vengono renderizzate ed elaborate interamente lato client.

## 6.3 Tecnologie e Tools

Per sviluppare l'applicazione è stato utilizzato l'IDE Open Source **Eclipse** nella versione 3.7. Di seguito saranno descritte brevemente le principali tecnologie utilizzate all'interno del sistema.

### Node.js ed Express.js

La tecnologia sulla quale si basa l'application server è Node.js e in questo modo il linguaggio principale sul quale si basa lo sviluppo di tutta l'applicazione è JavaScript. È stato utilizzato il framework Express.js per supportare lo sviluppo dell'application server grazie alle utili e robuste features da esso

offerte, le quali non mascherano alcuna funzionalità fornita da Node.js aprendo così le porte all'utilizzo di moduli per Node.js atti a supportare specifiche funionalità.

### TypeScript

Nonostante TypeScript sia ancora in fase developer preview, cioè in fase alpha, è già in uno stato utilizzabile ed è quindi possibile sfruttare le sue potenzialità utilizzando tale linguaggio in sostituzione di JavaScript. Quando si deve sviluppare un'applicazione di considerevole dimensione, è importante utilizzare i tools disponibili per agevolare lo sviluppo: tra i vari tools il più importante riguarda il *debugging* in quanto assiste lo sviluppatore nella risoluzione di eventuali errori. Attualmente non esistono debugger per il linguaggio TypeScript ma il compilatore tsc come è stato descritto nel capitolo precedente fornisce il supporto alla tecnologia Source Maps: utilizzando questa tecnologia è possibile debuggare direttamente sul codice sorgente TypeScript se il debugger JavaScript utilizzato fornisce il supporto a Source Maps.

Per quanto riguarda la *client application*, essa può essere debuggata in linguaggio JavaScript attraverso il debugger presente nella maggior parte dei browser: tra questi ad esempio Chrome fornisce il supporto alla tecnologia Source Maps, quindi lo sviluppo in TypeScript per quanto riguarda l'applicazione client è agevolato dalla possibilità di poter debuggare direttamente il codice prodotto.

Per quanto riguarda invece l'*application server*, esso può essere debuggato in linguaggio JavaScript utilizzando gli strumenti visti nel paragrafo 4.1.2, cioè il debugger integrato in Node.js, quelli integrato nel plugin per Node.js di Eclipse oppure utilizzando il modulo node-inspector. Nessuno di questi debugger però supporta la tecnologia Source Maps al momento, quindi il debug per il linguaggio TypeScript non è supportato. Sviluppi futuri di queste tre tecnologie però porteranno la compatibilità con Source Maps.

Considerati gli aspetti legati al debugging, per lo sviluppo della prima release di UpTennis è stata presa la decisione di utilizzare TypeScript solo per la client application, in quanto essa anche nella prima release presenta una discreta complessità; l'application server inizialmente presenta invece una complessità abbastanza contenuta e uno sviluppo iniziale in JavaScript comporterebbe, in caso di trasformazione futura in Typescript, un minimo

adattamento del codice. La contenuta complessità però è rilevata nella modularizzazione del codice, in quanto i singoli componenti (per lo più servizi REST) presentano ognuno una propria complessità in certi casi anche notevole; per questo motivo la possibilità di poter debuggare direttamente sul codice sorgente è importante. Per gli sviluppi futuri di UpTennis, in base alla disponibilità delle tecnlogie di debugging per TypeScript, se la complessità dell'application server crescerà si valuterà l'eventuale utilizzo di moduli Node.js che supportino lo sviluppo di unit-test per poter utilizzare TypeScript anche nello strato server.

La possibilità di effettuare debugging su codice sorgente, soprattutto in questa fase nella quale TypeScript si trova ancora nello stato di developer preview, assume una fondamentale importanza in quanto l'eventuale presenza di bug sul linguaggio, ad esempio utilizzando programmazione con unit-test, non garantisce allo sviluppatore che l'errore generato sia proprio oppure che sia dovuto appunto ad un bug di TypeScript.

### Backbone.js

La tecnologia Backbone.js descritta nel paragrafo 4.1.3 fornisce un livello di strutturazione dell'applicazione legato al design pattern MV*, il quale è stato utilizzato per la modellazione della client application. Inoltre fornisce il supporto alla gestione del routing client side, strumento molto utile soprattutto per lo sviluppo di applicazioni single page.

### JQuery

JQuery è la libreria JavaScript più diffusa per quanto riguarda l'elaborazione del DOM e l'utilizzo delle tecnologie AJAX ed è stata utilizzata nella client application. JQuery è anche utilizzata da altre librerie JavaScript le quali quindi ne sono strettamente dipendenti: tra queste troviamo anche Backbone.js, la quale ne ha una dipendenza indiretta in quanto dipendendo da Underscore.js (libreria JS di utility) ne eredita le dipendenze tra le quali è presente JQuery.

### Bootstrap

Bootstrap è una libreria JavaScript rilasciata dal team di sviluppo di Twitter che agevola la realizzazione dell'interfaccia grafica di una client applica-

tion, attraverso un set di componenti già pronti all'utilizzo e soprattutto ad un sistema di layouting che evita allo sviluppatore di dover investire tempo prezioso per ordinare i componenti presenti all'interno dell'interfaccia grafica.

### MongoDB e Mongoose.js

Per la memorizzazione dei dati su store persistente è stata effettuata la scelta di utilizzare MongoDB, un database documentale (NO-SQL) dalle alte performance, scalabile, che memorizza i dati in documenti JSON, facilitando la mappatura dei tipi di dato dei valori in essi contenuti con i tipi di dato di JavaScript. Per interfacciare l'application server con MongoDB sono disponibili diversi progetti open-source ed in questo caso è stato scelto di utilizzare Mongoose.js, attualmente il più diffuso.

### Async.js e Underscore.js

Sia lato client che lato server sono state utilizzate librerie di utilità al supporto della programmazione JavaScript: Async.js e Underscore.js; la prima supporta la programmazione asincrona fornendo API che permettono un maggiore controllo e una maggiore leggibilità del codice necessario per effettuare operazioni asincrone in serie, in parallelo, ecc., mentre la seconda fornisce un ricco set di API che facilitano le operazioni più comuni ad esempio su array, oggetti, funzioni, ecc.

### Require.js

Mentre Node.js offre a livello di piattaforma la modularizzazione del codice implementando il pattern CommonJS, per la client application è necessario utilizzare una tecnologia di terze parti ed è stato optato per l'utilizzo di Require.js in quanto fornisce anche il supporto alla gestione delle dipendenze tra librerie ed all'ottimizzazione/minimizzazione del codice JavaScript finale che sarà effettivamente eseguito dai browser. TypeScript come è stato illustrato nel capitolo precedente fornisce il supporto per entrambi i pattern di modularizzazione a livello di compilazione.

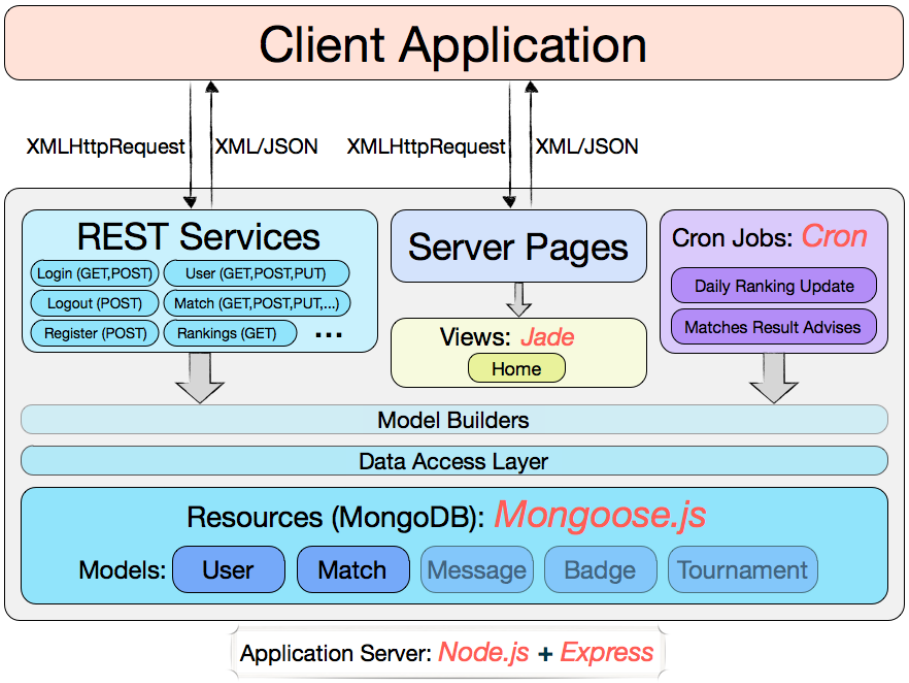


Figura 6.2: Architettura UpTennis - Application Server

## 6.4 Application Server

Nonostante l'application server (architettura descritta in figura 6.3) non sia stato attualmente sviluppato utilizzando TypeScript, nelle future release di UpTennis sarà sicuramente effettuata la conversione dei sorgenti attualmente sviluppati in JavaScript e quindi in questo paragrafo saranno analizzati i cambiamenti che dovranno essere apportati al codice sorgente per essere compatibile con il processo di compilazione; questo processo ha una rilevanza importante in quanto, essendo TypeScript un superset di JavaScript compatibile con la sua sintassi al 100%, ogni applicazione sviluppata in JavaScript può essere convertita in TypeScript per sfruttare le sue potenzialità; un progetto convertito in TypeScript inoltre potrà godere dei benefici della programmazione object oriented e quindi successivamente al processo di conversione, può essere attuato un piano di ristrutturazione del codice atto a migliorarne la qualità e le potenzialità.

### 6.4.1 Conversione in TypeScript

Il primo passo fondamentale di conversione di sorgenti da JavaScript a TypeScript è la sostituzione dell'estensione `.js` in `.ts` in ogni file/modulo. Nel caso di sorgenti scritti in codice JavaScript puro, senza l'utilizzo di librerie di terze parti, con questa semplice modifica la conversione effettuata manderebbe la compilazione dei nuovi sorgenti TypeScript a buon fine; ovviamente senza l'utilizzo di librerie di terze parti, ogni sorgente JavaScript essendo indipendente, dopo la conversione implicherebbe la compilazione di ogni singolo file TypeScript: vedremo quindi che anche una singola libreria di gestione della modularizzazione di codice JavaScript, comporta delle operazioni da effettuare per passare il processo di compilazione con esito positivo. Nel precedente capitolo analizzando TypeScript è stato affrontato il problema legato all'utilizzo di librerie JavaScript di terze parti all'interno di un sorgente TypeScript: è stato sottolineato come e perché sia necessario utilizzare per ognuna di esse un file di dichiarazione della propria interfaccia.

**Definizione delle librerie** L'application server è realizzato in Node.js e come microframework di supporto allo sviluppo è stato utilizzato Express, quindi per entrambe le tecnologie è necessario utilizzare un file d.ts che ne descriva l'interfaccia. Attualmente per Eclipse non sono disponibili tools al supporto della programmazione in TypeScript, quindi l'utilizzo di tali file d.ts è necessario solamente per poter eseguire il processo di compilazione correttamente. Come avviene per Node.js ed Express, anche per qualsiasi altra libreria di terze parti utilizzata nell'application server sarà necessario utilizzare un corrispondente file di dichiarazione. Tali file di dichiarazione devono essere inclusi nell'intestazione del sorgente TypeScript nel quale vengono utilizzati i corrispondenti moduli. È importante sottolineare come il compilatore TypeScript consideri i file di definizione contenuti all'interno dei sorgenti TypeScript: quando si compila un'applicazione strutturata in moduli il compilatore in fronte alla prima importazione di modulo che incontra, carica in memoria i file di dichiarazione in esso presenti e se tra di questi ce ne sono alcuni già caricati precedentemente, non effettua nuovamente il caricamento. È buona norma però nell'intestazione di un modulo specificare sempre tutte le inclusioni dei file di definizione dei moduli utilizzati all'interno di esso, perché in caso di mancata definizione, se nei moduli caricati

precedentemente dal compilatore non compaiono tali definizioni, si avranno errori di compilazione in corrispondenza dell'utilizzo di tali moduli.

**Modularizzazione** Il secondo aspetto importante da considerare per la conversione di un application server sviluppato in Node.js è la modularizzazione del codice sorgente: all'interno di un file sorgente TypeScript per utilizzare un modulo esterno è necessario utilizzare l'istruzione `import`, mentre Node.js utilizza il pattern CommonsJS che richiede invece l'utilizzo dell'istruzione require. La conversione richiede quindi che in ogni sorgente JavaScript Node.js da trasformare in TypeScript, sia utilizzata l'istruzione import in sostituzione dell'istruzione require ogni qualvolta sia necessario utilizzare un modulo esterno. Di seguito è mostrato un esempio di come cambia l'importazione del modulo Express utilizzato all'interno dell'applicazione Node.js:

```
//JavaScript
var express = require('express')

//TypeScript
import express = module("express");
```

Considerati questi passaggi da effettuare per sviluppare un'applicazione Node.js in TypeScript, la restante parte di sviluppo è strettamente dipendente da come si vogliono sfruttare le potenzialità del linguaggio TypeScript, cioè la possibilità di utilizzare costrutti di programmazione object oriented con tipizzazione statica opzionale. Esempi di questo genere potranno essere affrontati in dettaglio analizzando lo sviluppo della client application, interamente sviluppata con TypeScript.

## 6.5 Client Application

Per quanto riguarda lo sviluppo della client application (architettura descritta in figura 6.4), si è partiti fin dall'inizio a sviluppare in TypeScript in quanto, nonostante sia ancora in fase alpha e presenti diversi bug che prontamente vengono affrontati e risolti, è un linguaggio già utilizzabile per sfruttare le principali caratteristiche di un linguaggio di programmazione object oriented; in aggiunta per la programmazione client side sono presenti strumenti

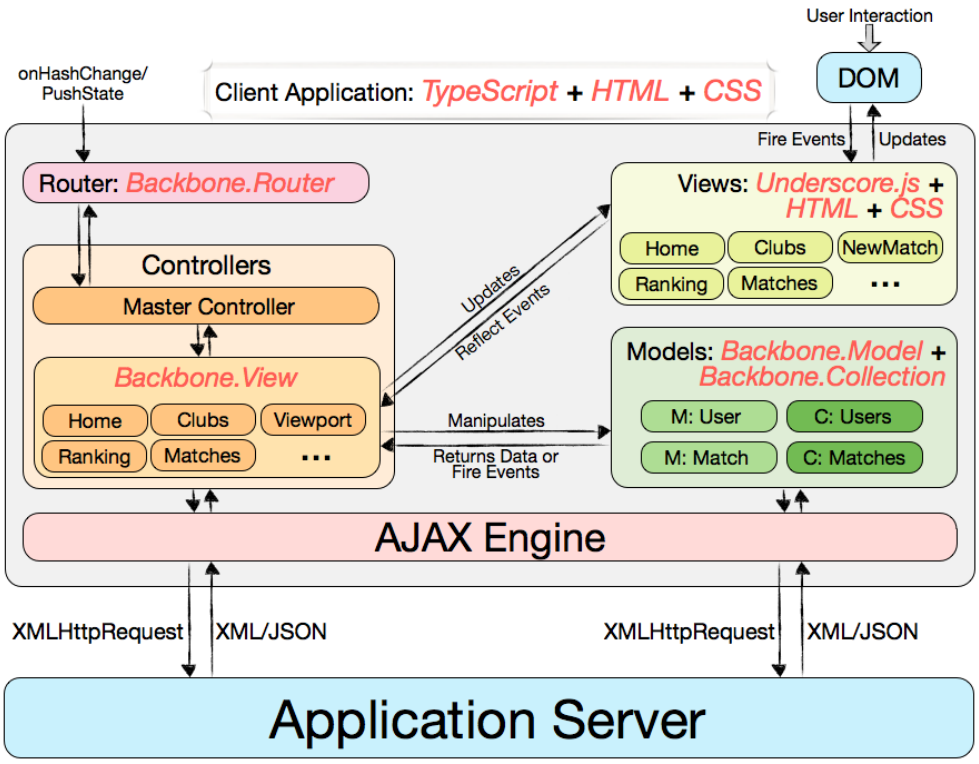


Figura 6.3: Architettura UpTennis - Client Application

di debugging che utilizzano la tecnologia *Source Maps* e quindi permettono il debug direttamente su codice sorgente scritto in TypeScript: questo aspetto è di particolare rilevanza in quanto la client application anche nella prima release di UpTennis è caratterizzata da una discreta complessità ed un approccio simile a quello utilizzato per l'application server comporterebbe tempi di conversione sicuramente non trascurabili.

In questa sezione saranno affrontati tutti i casi particolari di utilizzo del linguaggio TypeScript per quanto riguarda lo sviluppo della client application, sottolineando le differenze rispetto ad un approccio allo sviluppo basato completamente su linguaggio JavaScript; sarà descritto ogni singolo caso particolare anche attraverso alcuni esempi di codice.

### 6.5.1 Utilizzo di tools

TypeScript è un linguaggio sviluppato da Microsoft e, a differenza dai prodotti distribuiti sul mercato precedentemente sempre dalla stessa azienda, è un progetto open-source; Microsoft attorno al mercato dello sviluppo del software ha sempre mostrato particolare attenzione all'aspetto del tooling, dotando gli sviluppatori di IDE e linguaggi accompagnati da tools che sicuramente aiutano lo sviluppatore nella creazione del codice sorgente di un progetto. Per quanto riguarda TypeScript, restando in linea con le precedenti esperienze, Microsoft fin dal principio ha rilasciato insieme al linguaggio anche in questo caso un set di utili tools (syntax highlighting, error checking ed intellisense) sotto-forma di plugin per l'IDE Visual Studio 2012 e per altri (ancora molto pochi) IDE come Vim, Sublime Text, Emacs e Cloud9; tra questi IDE però per motivi di diversa natura (non open-source, non compatibilità con altre tecnologie utili allo sviluppo, ecc.), ancora non compaiono IDE open-source tra i più utilizzati per lo sviluppo, come ad esempio Eclipse: la possibilità di sviluppare tools direttamente in linguaggio JavaScript (come ha fatto direttamente Microsoft con lo strumento *Play* presente sul sito ufficiale di TypeScript) apre però le porte ad una futura rapida diffusione di tools anche per questi IDE. L'utilizzo di tools agevola sicuramente lo sviluppo, ma non è di certo la caratteristica innovativa di TypeScript, in quanto gli aspetti più rilevanti sono offerti dal linguaggio stesso a livello di features; nonostante ciò, TypeScript ha comunque colmato un gap che JavaScript non sarebbe altrimenti riuscito a superare, in quanto ad esempio un tools molto utile come l'*intellisense*, in un linguaggio con tipizzazione dinamica non potrebbe esistere.

### 6.5.2 Tecnologie utilizzate

Come è stato già descritto per l'application server, per ogni libreria JavaScript di terze parti che viene utilizzata all'interno di un'applicazione TypeScript, è necessario importare il file `d.ts` contenente la sua definizione almeno prima del primo utilizzo (meglio importarli comunque in un punto dei sorgenti seguendo un criterio ordinato): l'importazione avviene sempre utilizzando i *reference comments*.

### Require.js

Prima di passare alla descrizione della strutturazione vera e propria del codice TypeScript, sarà descritta la gestione della modularizzazione dei sorgenti. Siccome si è vincolati ad utilizzare la sintassi proposta da TypeScript, la scelta deve ricadere su una delle due possibilità che quest'ultimo mette a disposizione a livello di generazione del codice JavaScript post-compilazione: mentre per l'application server si è obbligati all'utilizzo del pattern CommonsJS, per la client application si può scegliere liberamente tra le due possibilità; la scelta è ricaduta sul pattern AMD, implementato dalla nota libreria *Require.js*, la quale oltre alla gestione della modularizzazione offre la possibilità di definire dipendenze tra librerie e di ottimizzare/minimizzare il codice JavaScript generato.

Attraverso l'utilizzo di questa libreria sarà quindi definito il punto d'accesso della client application, preceduto dalla dichiarazione delle librerie di terze parti utilizzate (e da eventuali dipendenze tra di esse) e dall'importazione dei loro file `d.ts`; il punto d'accesso dell'applicazione è buona norma separarlo dalle configurazioni delle librerie in un file sorgente differente: per fare ciò è necessario creare il nostro primo modulo che chiameremo `AppMain`, corrispondente quindi al punto d'accesso dell'applicazione. Il file che conterrà le configurazioni delle librerie (`AppConfig`) sarà a sua volta un modulo TypeScript, quello sul quale sarà avviato il processo di compilazione; all'interno di tale modulo dovrà essere presente la dichiarazione di importazione del modulo contenente il punto d'accesso dell'applicazione, ma solo in questo caso sarà utilizzata la modalità fornita dalla libreria Require.js in modo tale che su di esso possano essere gestite le dipendenze dalle librerie di terze parti utilizzate; di seguito sarà mostrato il contenuto parziale del file `AppConfig` lasciando solamente visibile la gestione della dipendenza dalla libreria JQuery:

```
//Typescript Type Definitions imports
/// <reference path="./modules/require.d.ts" />
/// <reference path="./modules/jquery.d.ts" />
/// <reference path="./app/AppMain.ts" />

require.config({
    baseUrl: '/',
    paths: {
        'jquery': 'lib/jquery/jquery',
```

```
        ...
        'main': 'app/AppMain'
    },
    shim: {
        jquery: {
            exports: '$'
        },
        ...
        'main': {
            deps: [
                "jquery",
                ...
            ],
            exports: 'main'
        }
    }
});

require(['main'],
    (main) => {
    $(() => {
        var appMain = new main.AppMain();
        appMain.run();
    });
});
```

Senza entrare nel dettaglio del funzionamento della libreria Require.js, in questo modulo vengono utilizzate due funzioni di tale libreria: la prima (`require.config`) permette la definizione delle librerie da utilizzare e la gestione delle eventuali dipendenze, mentre la seconda (`require`) è la vera e propria importazione del modulo `AppMain` che può essere utilizzato all'interno dello scope della funzione richiamata come callback in seguito all'avvenuto caricamento di tutte le librerie dipendenti dal modulo main. Il modulo `AppMain` è però utilizzabile dal punto di vista di TypeScript all'interno del modulo di configurazione (`AppConfig`) solamente se viene importata la sua definizione in cima al file, altrimenti il compilatore genererà un errore: questo avviene perché per la sua importazione non è stato utilizzato il meccanismo implementato sul compilatore TypeScript.

A questo punto la client application sarà in grado di iniziare la sua esecuzione a partire dal metodo `run()` eseguito sull'istanza della classe TypeScript `AppMain` definita all'interno del modulo `AppMain` stesso.

Sul browser sarà eseguito però il codice JavaScript generato dal compilatore TypeScript, quindi sarà necessario importare nel file HTML corrispondente alla single page dell'application lo script tag contenente il riferimento al primo file JavaScript da eseguire: l'utilizzo della libreria Require.js impone che venga utilizzato lo script tag nel seguente modo:

```
<script data-main="/AppConfig" type="text/javascript"
src="/lib/require/require.js">
```

Tale tag va inserito nel blocco HTML head e carica la libreria Require.js alla quale viene passato il riferimento al file JavaScript contenente le sue configurazioni e il punto di start dell'applicazione.

### Backbone.js

Nel punto d'accesso dell'applicazione saranno definiti e utilizzati i componenti dell'applicazione in base agli scenari di utilizzo che si verificheranno. La strutturazione della client application è stata effettuata avvalendosi dell'utilizzo della libreria Backbone.js descritta nel paragrafo 4.1.3, quindi come primo passo è necessario definire all'interno delle configurazioni di Require.js le sue dipendenze da eventuali altre librerie (Underscore.js e JQuery) e le corrispondenti definizioni TypeScript (`d.ts`). La client application è stata quindi strutturata dividendola concettualmente nei seguenti componenti:

- *Controller*: è un componente che estende la classe `Backbone.Events` per modellare la comunicazione con altri componenti dell'applicazione utilizzando l'approccio a scambio di messaggi; i controller contengono solamente la business logic della client application.
- *Router*: è il componente che gestisce la navigazione della client application attraverso l'ascolto dei cambiamenti dell'url generati dall'utilizzo di link presenti all'interno delle viste; estende la classe `Backbone.Router`.
- *Model*: è il componente che modella le entità del sistema e si occupa delle funzionalità di sincronizzazione con l'application server; estende la classe `Backbone.Model`.

- *Collection*: è il componente che gestisce insiemi di istanze di modelli; estende la classe `Backbone.Collection`.
- *View*: è il componente che si occupa della business logic legata alle viste della client application e della loro renderizzazione; estende la classe `Backbone.View`.
- *Template*: è il componente che definisce solamente il contenuto di una vista utilizzando il linguaggio di templating Underscore integrato all'interno di Backbone.

Ognuno di questi componenti è strutturato utilizzando i concetti della programmazione object oriented e quindi è mappato con una corrispondente classe TypeScript, la quale estendendo da un componente Backbone, ne eredita tutte le sue caratteristiche definite nel file di definizione della libreria (`d.ts`), alle quali ne potranno essere aggiunte ulteriori in base alle necessità.

Di seguito sarà mostrato il codice corrispondente alla classe che identifica l'entità di modello Match, la quale estenderà quindi dalla classe Backbone.Model:

```
export class Match extends Backbone.Model{

    //Class Fields

    //Class Methods

    public addPlayer(attributes?: any, options?: any){
        $.ajax({
            type: "POST",
            url: "/api/match/player",
            context: this,
            data: {
                matchId: this.get('matchId'),
                userId: attributes.userId,
                teamId: attributes.teamId
            }
        }).done(function(response){
            this.clear();
```

```
            this.set(response.match);
            if(options.success)
                options.success(this,response);
        }).fail(options.error);
    }

    public removePlayer(attributes?: any, options?: any){
        $.ajax({
            type: "DELETE",
            url: "/api/match/player",
            context: this,
            data: {
                matchId: this.get('matchId'),
                userId: attributes.userId
            }
        }).done(function(response){
            this.clear();
            this.set(response.match)
            if(options.success)
                options.success(this,response);
        }).fail(options.error);
    }

    public setScore(attributes?: any, options?: any){
        $.ajax({
            type: "POST",
            url: "/api/match/score",
            context: this,
            data: {
                matchId: this.get('matchId'),
                score: attributes.score
            }
        }).done(function(response){
            this.clear();
            this.set(response.match)
            if(options.success)
                options.success(this,response);
```

```
        }).fail(options.error);
    }

    public confirmScore(options?: any){
        $.ajax({
            type: "PUT",
            url: "/api/match/score",
            context: this,
            data: {
                matchId: this.get('matchId')
            }
        }).done(function(response){
            this.clear();
            this.set(response.match)
            if(options.success)
                options.success(this,response);
        }).fail(options.error);
    }

    public addUserInvitation(attributes?: any, options?: any){
        $.ajax({
            type: "POST",
            url: "/api/match/invitation",
            context: this,
            data: {
                matchId: this.get('matchId'),
                userId: attributes.userId
            }
        }).done(function(response){
            this.clear();
            this.set(response.match)
            if(options.success)
                options.success(this,response);
        }).fail(options.error);
    }

    // Backbone Methods
```

```
    initialize() {
        this.urlRoot = '/api/matches';
    }

    save(attributes?: any, options?: JQueryAjaxSettings){
        var successCallback = options.success;
        options.success = (data: any, textStatus: any, jqXHR:JQueryXHR)
            => {
                data.clear();
                data.set(textStatus.match);
                if(successCallback)
                    successCallback(data, textStatus, jqXHR);
            };
        super.save(attributes,options);
    }
}
```

Ogni file TypeScript contenente la definizione di uno di questi componenti è a sua volta un modulo e quindi una classe definita al suo interno per poter essere utilizzata al di fuori dal modulo stesso dovrà essere preceduta dalla direttiva export; di conseguenza all'interno di un modulo per poterne utilizzare altri definiti esternamente basterà utilizzare il costrutto import.

Di seguito sarà mostrato il codice corrispondente all'aspetto appena descritto relativamente alla classe Viewport, una vista (Backbone.View) gestita dal Master controller che a sua volta gestisce e compone tutte le viste presenti nell'applicazione:

```
import mn = module("app/views/header/MainNavBar");
import f = module("app/views/Footer");
import hs = module("app/views/sections/Home");
import rs = module("app/views/sections/Ranking");
...
import bv = module("app/views/BaseView");

export class Viewport extends bv.BaseView{
    ...
}
```

### 6.5.3 Object Oriented Programming

I componenti che caratterizzano il sistema realizzato rappresentano quindi le classi, le quali a loro volta sono definite ciascuna in un modulo. Per ogni classe saranno definiti attributi e metodi che ne definiranno rispettivamente stato e comportamento e per ciascuno di essi possono essere definite regole di visibilità (public o private); eventualmente possono essere definiti attributi o metodi statici utilizzando la direttiva static posta dopo l'eventuale visibilità.

Gli *attributi* possono essere definiti utilizzando opzionalmente un tipo di dato primitivo o un tipo di dato rappresentato da una classe custom; nel caso di assegnamento di un tipo di dato il valore assegnato a tale attributo all'interno dei metodi della classe deve sempre rispettare tale signature altrimenti il compilatore genererà un errore a compile-time.

All'interno di ogni metodo saranno eseguite istruzioni che definiranno il comportamento della classe e all'interno di esse lo scope corrisponderà proprio alla classe stessa, potendo agire sugli attributi e potendo utilizzare i metodi disponibili attraverso l'utilizzo dell'oggetto `this`. Bisogna fare particolare attenzione nel caso di utilizzo di callback per gestire operazioni asincrone, in quanto lo *scope* interno alla callback utilizzando la sintassi di JavaScript non viene preservato; le possibili soluzioni per poter preservare lo scope sono due: la prima sfruttando il meccanismo JavaScript di visibilità dello scope di funzioni innestate tra loro, che permette di utilizzare all'interno di una funzione anche le variabili definite esternamente; basterà quindi creare all'interno di un metodo di una classe una variabile che contiene il riferimento all'oggetto `this` e successivamente utilizzare tale variabile all'interno delle callback definite dentro tale metodo. La seconda soluzione invece utilizza direttamente un costrutto alternativo di Typescript per creare le funzioni (*arrow function expression*) che, utilizzato per creare le callback, permette di preservare lo scope al loro interno; questa soluzione è più sicura in quanto non c'è il rischio di incorrere in errori legati alla sovrascrittura del valore della variabile utilizzata per referenziare l'oggetto `this`. Di seguito saranno esposte con un esempio le due soluzioni alternative, poste rispettivamente nei metodi `sol1` e `sol2` della classe `Test`:

```
class Test{
```

```
    private a :string;

    private sol1(){
        var _this = this;
        asyncOperation( function(param :string){
            _this.a = param;
        });
    }

    private sol2(){
        asyncOperation( (param :string) => {
            this.a = param;
        });
    }
}
```

TypeScript permette la realizzazione di ereditarietà tra classi in modo molto semplice e di seguito sarà mostrato un esempio di codice relativo ad una classe che rappresenta un controller della vista della sezione Home di UpTennis; tale controller dovrà estendere dalla classe Backbone.View per ereditarne le caratteristiche e in aggiunta sono stati utilizzati due ulteriori livelli di ereditarietà tra classi, per gestire la condivisione di caratteristiche e funzionalità tra componenti simili all'interno dell'applicazione: la classe Home estenderà quindi la classe AppSection, la quale estenderà la classe BaseView che infine estende la classe Backbone.View; dalla classe AppSection estenderanno quindi tutte le viste che rappresentano una sezione dell'applicazione (quindi la classe Viewport vista in precedenza non estenderà da quest'ultima), mentre dalla classe BaseView estenderanno tutte le viste dell'applicazione: in tali classi saranno contenute ovviamente caratteristiche e funzionalità comuni tra le classi che da esse estenderanno.

```
... [app/views/BaseView.ts] ...

export class BaseView extends Backbone.View{
    // Class Fields
    loader: any;

    // Class Methods
```

```
    public showAlert(alertContainer: any, msgTxt: string,
        alertType?: string = '', closable?: bool = true,
        fadeOut?: bool = true, timeout?: number = 5000){
            if(alertContainer){
                alertContainer.append('<div class="alert '+alertType+'">'+
                    ((closable) ? '<button type="button" class="close" ' +
                    'data-dismiss="alert">&times;</button>' : '') +
                    msgTxt +'</div>');
                if(fadeOut)
                    alertContainer.find('.alert').delay(timeout)
                        .fadeOut("slow");
            }
    }
}

... [app/views/sections/AppSection.ts] ...

import bv = module("app/views/BaseView");

export class AppSection extends bv.BaseView{
    // Class Fields
    userLogged: any;
    userLoggedTemplate: string;
    type: string;

    // Class Methods
    constructor(){ super(); }

    public setUserLoggedIn(userData: any){
        this.updateUserLoggedInData(userData);
        if(this.type == "private"){
            require([this.userLoggedTemplate], (t) => {
                this.template = _.template(t);
                this.render();
            });
        }
        else{
```

```
            this.render();
        }
    }

    public updateUserLoggedInData(userData: any){
        this.userLogged = userData;
    }
    ...

    //Backbone Methods
    initialize() {
        this.type = "private";
        this.loader = $("#app-loading-spinner");
        this.userLoggedTemplate =
            'text!app/templates/sections/PrivateSection.html';

        _.bindAll(this, 'beforeRender', 'render', 'afterRender');
        this.render = _.wrap(this.render, (render) => {
            this.beforeRender();
            render();
            this.afterRender();
            return this;
        });

        this.afterInitialize();
    }

    afterInitialize(){
       ...
    }

    beforeRender(){ }

    render(){
        ...
    }
```

```
    afterRender(){ }
}

... [app/views/sections/Home.ts] ...

import as = module("app/views/sections/AppSection");
import mc = module("app/collections/Match");
import um = module("app/models/User");

export class Home extends as.AppSection{
    // Class Fields
    private userLogged: any;
    private userToRender: any;
    private userToRenderMatches: any;
    private userProfileMode: Boolean;
    private changeUserPictureTooltip: any;
    private dataTables: any;

    // Class Methods

    constructor(){ super(); }

    //Override
    public setUserLoggedIn(userData: any){
        this.userLogged = userData;
        if(this.userProfileMode)
            this.showUserProfile();
        else
            this.showUserProfile(userData);
    }

    public showUserProfile(userProfileData?: any){
        ...
    }

    public showHomePage(){
        if(this.userLogged){
```

```
            if(this.userLogged.get('status') == "confirmed"){
                this.trigger('goToUrl',this.userLogged.get('linkname'),true);
                return;
            }
        }
        if(this.userProfileMode){
            this.userProfileMode = false;
            require(['text!app/templates/sections/Home.html'], (t) => {
                this.template = _.template(t);
                this.userProfileMode = false;
                this.render();
            });
        }
    }

    ...

    // Backbone Methods

    private afterInitialize() {
        this.type = "public";

        this.dataTables = [];
        this.tagName = "div";
        this.$el = $('#home-section');
        this.events = {
            'mouseenter #userPicture': 'handleUserPictureOver',
            'mouseleave #userPicture': 'handleUserPictureOut',
            'mouseleave #userPictureContainer .tooltip':
                'handleChangeUserPictureTooltipOut',
            'click #btnChangeUserPicture':
                'handleBtnChangeUserPictureClick',
            'change #userPictureUploader input':
                'handleUserPictureUpload',
            'click #userProfileMainNavTabs li a':
                'handleUserProfileMainNavTabsClick',
            'click #userProfileMainNavTabs table.matches-table tbody tr':
```

```
                'handleMatchClick'
        };
        super.afterInitialize();
        this.userLogged = false;
        this.userProfileMode = true;
        this.showHomePage();
    }

    ...

    // Event Handlers

    private handleUserPictureOver(e){
        ...
    }

    private handleUserPictureOut(e){
        ...
    }

    ....
}
```

La precedente porzione di codice mostra come utilizzare in TypeScript la maggior parte dei costrutti a disposizione dello sviluppatore, dalla creazione di classi, alla realizzazione di ereditarietà all'utilizzo dell'optional static typing; di seguito sarà mostrato invece come il compilatore TypeScript converte tale codice in JavaScript e tale codice corrisponderà quindi a quanto lo sviluppatore avrebbe dovuto realizzare: tale codice sarà riassunto mostrando solamente la classe Home nella sua struttura principale.

```
var __extends = this.__extends || function (d, b) {
    function __() { this.constructor = d; }
    __.prototype = b.prototype;
    d.prototype = new __();
};
define(["require", "exports", "app/views/sections/AppSection",
```

```
"app/collections/Match", "app/models/User"], function(require,
exports, __as__, __mc__, __um__) {
var as = __as__;
var mc = __mc__;
var um = __um__;

var Home = (function (_super) {
    __extends(Home, _super);
    function Home() {
        _super.call(this);
    }
    Home.prototype.setUserLoggedIn = function (userData) {
        ...
    }

    ...

    Home.prototype.showHomePage = function () {
        var _this = this;
        if(this.userLogged) {
            if(this.userLogged.get('status') == "confirmed") {
                this.trigger('goToUrl',
                    this.userLogged.get('linkname'), true);
                return;
            }
        }
        if(this.userProfileMode) {
            this.userProfileMode = false;
            require([
                'text!app/templates/sections/Home.html'
            ], function (t) {
                _this.template = _.template(t);
                _this.userProfileMode = false;
                _this.render();
            });
        }
    };
```

```
        ...

        return Home;
    })(as.AppSection);
    exports.Home = Home;
})
```

Si può notare quando il codice TypeScript risulti più leggibile data la più concisa sintassi necessaria per ottenere lo stesso risultato; inoltre data la natura di JavaScript i concetti mancanti rispetto a TypeScript non sono presenti.

Definita la strutturazione del codice attraverso l'utilizzo dei costrutti offerti dall'OOP, la restante parte di sviluppo è strettamente legata alla business logic dell'applicazione applicando ove necessario l'utilizzo di librerie di terze parti in aggiunta ai costrutti base di JavaScript.

# Capitolo 7

# Conclusioni

JavaScript ricopre al giorno d'oggi un ruolo fondamentale sia nello sviluppo di applicazioni web, che nello sviluppo di applicazioni per piattaforme desktop e mobile. Realizzare progetti di medie/grandi dimensioni con JavaScript vuol dire però spesso ottenere un codice caotico e difficilmente mantenibile.

Negli ultimi anni sono nati progetti atti a risolvere i problemi insiti in JavaScript e tra questi i più rilevanti sono TypeScript e Dart. Entrambi offrono diverse opportunità agli sviluppatori, i quali possono ottenere in modo semplice codice estremamente leggibile, mantenibile e strutturato con il vantaggio di avere il controllo dei tipi a compile-time. Sia Dart che TypeScript si pongono quindi simili obiettivi e sono entrambi progetti importanti e ben realizzati, ma la differenza più grande tra loro è che si pongono sul mercato esattamente nel modo opposto: Dart è sicuramente un progetto più ambizioso rispetto a TypeScript, ma la sua natura lo limita in partenza per quanto riguarda le possibilità di utilizzo, in quanto non essendo ancora supportato nativamente nei browser più diffusi (attualmente lo è solo su Chrome), non è possibile trarne a pieno i benefici che esso offre; TypeScript invece sfrutta questo aspetto nel modo esattamente opposto, cioè mantenendosi compatibile con la sintassi e la semantica del linguaggio JavaScript, aggiungendo un set di feature che lo trasformano in linguaggio più adatto per lo sviluppo di applicazioni strutturate.

Il vantaggio di Dart è quindi legato alle migliori performance che esso offre all'interno del proprio runtime environment ed alla solidità che un linguaggio progettato ex-novo può avere rispetto ad un linguaggio datato che ha subito negli anni enormi cambiamenti ed evoluzioni; per quanto riguarda Ty-

peScript invece preservando sintassi, semantica e performance di JavaScript, il vantaggio è legato alla quasi nulla curva di apprendimento del linguaggio per la numerosa comunità di sviluppo legata al mondo JavaScript, ma soprattutto la compatibilità con le numerose librerie esistenti sul web.

Entrambi i progetti offrono la possibilità di ottenere codice JavaScript a partire da sorgenti realizzati con i loro relativi linguaggi attraverso un processo di compilazione: l'output garantisce quindi la corretta esecuzione su qualunque browser e piattaforma.

Fino a pochi anni fa JavaScript era un linguaggio utilizzato solamente per la programmazione di applicazioni web client, mentre negli ultimi anni grazie a tecnologie come Node.js, è possibile utilizzare JavaScript come linguaggio per realizzare Web Server: anche in questo caso TypeScript è in grado di aggiungere un livello ulteriore di solidità ad una nuova soluzione che ha portato omogeneità nello sviluppo di applicazioni Web nel loro complesso. Il concetto di riusabilità del codice potrà essere così sfruttato per componenti comuni inter strato applicativo oltre che ai classici casi intra strato applicativo o inter applicazione.

Le funzionalità che TypeScript aggiunge a JavaScript non sono molte, ma producono grandi vantaggi agli sviluppatori che sono abituati a simili caratteristiche nei linguaggi che utilizzano per lo sviluppo di applicazioni strutturate.

Per quanto riguarda la realizzazione di sistemi concorrenti, TypeScript a livello di linguaggio non offre nessun supporto aggiuntivo a quanto è offerto da JavaScript e quindi essi possono essere realizzati utilizzando i Web Worker; questo aspetto va leggermente a vantaggio di Dart, il quale offre a livello di linguaggio il concetto di Isolate, attraverso il quale possono essere realizzati gli stessi sistemi aggiungendo un grado di modellazione esteso rispetto ai Web Worker.

TypeScript non aggiunge nemmeno features a supporto della programmazione asincrona, caratteristica delicata di JavaScript che se mal utilizzata può generare scenari applicativi non desiderati; bisogna quindi appoggiarsi a delle librerie JavaScript al supporto della programmazione asincrona come ad esempio Async.js che agevolano lo sviluppo di casi critici, ma che comunque non evitano di incorrere in eventuali errori di programmazione (conseguenza di una difficoltà non banale nell'affrontare questi tipi di problemi).

La possibilità di avere a disposizione tool a supporto del linguaggio è una caratteristica aggiuntiva che TypeScript fornisce a differenza di JavaScript e

comporta vantaggi dal punto di vista dell'agilità di sviluppo.

Nonostante TypeScript sia ancora in fase alpha e sottoposto a continui rilasci di aggiornamenti, si trova già in una fase avanzata a tal punto da poter essere utilizzato per realizzare applicazioni o per convertirne delle già esistenti scritte in JavaScript. Per un primo approccio al linguaggio, è disponibile il sito ufficiale [12], ricco di spunti, esempi pratici e una documentazione tecnica molto approfondita, mentre il progetto open source è ospitato su CodePlex e seguito da una comunità molto attiva che segnala costantemente bug, dubbi e proposte future.

Nei suoi primi mesi di vita TypeScript ha attirato su di se l'attenzione anche dei personaggi più importanti legati a JavaScript come ad esempio Douglas Crockford, il quale ha dichiarato che TypeScript è il miglior front-end JavaScript tra quelli esistenti. Sicuramente è un buon strumento per avvicinarsi in anticipo all'armonia proposta dallo standard ECMAScript 6, ma bisogna considerare che questo standard è ancora soggetto a possibili cambiamenti. TypeScript è ancora in fase di developer preview, ma è legittimo chiedersi cosa potrebbe eventualmente accadere nel caso le specifiche ECMAScript 6 dovessero cambiare: Microsoft dovrà prendere una decisione riguardo la possibilità di adeguarsi ai cambiamenti oppure di mantenere le attuali specifiche. Ci sono quindi ancora ambiguità legate a TypeScript, ma sicuramente vale la pena seguirne le evoluzioni. Utilizzarlo per sviluppare un progetto di importanti dimensioni potrebbe essere un'arma a doppio taglio, in quanto possibili drastici cambiamenti all'interno del linguaggio potrebbero implicare un consistente impiego di tempo per adattare il progetto.

Come caso di studio è stato affrontato lo sviluppo di un sistema di medie dimensioni con programmate evoluzioni future ed il risultato conseguito è stata la produzione di un codice sicuramente molto strutturato e con un livello di leggibilità maggiore rispetto al codice JavaScript che altrimenti sarebbe dovuto essere stato scritto; anche in casi di manutenzione del codice il processo è sempre stato agile grazie alla separazione dei componenti sfruttando la programmazione Object Oriented. Un altro vantaggio da sottolineare è il rilevamento di errori a compile-time che ha sicuramente fatto risparmiare tempo per l'individuazione e la correzione; il tutto a discapito della perdita di una delle caratteristiche più amate di JavaScript, cioè la natura di linguaggio interpretato che conseguiva uno sviluppo più agile data la mancanza di un processo di compilazione. Per il supporto allo sviluppo non sono stati utilizzati tools data l'attuale disponibilità solo per pochi ambienti di sviluppo, ma

in un futuro non molto lontano anche questo aspetto avrà la sua rilevanza nel portare TypeScript al successo.

Infine è importante sottolineare come le tecnologie Web in generale stiano riscontrando negli ultimi tempi un alto interesse anche per quanto riguarda lo sviluppo per altre piattaforme diverse da quella Web: questo aspetto renderà ancora più necessaria la presenza di un linguaggio solido e idoneo per lo sviluppo di applicazioni altamente strutturate. Avvalendosi di metodologie di ingegnerizzazione del software come il Model-Driven Development si potranno generare applicazioni multi-piattaforma risparmiando una considerevole quantità di tempo e di conseguenza diminuiranno anche i costi di produzione.

# Bibliografia


[1] Wikipedia. *http://it.wikipedia.org*

[2] Tim O'Reilly: *What Is Web 2.0*
*http://oreilly.com/web2/archive/what-is-web-20.html*

[3] Jason Farrell, George S. Nezlek: *Rich Internet Applications: The Next Stage of Application Development.*

[4] James Governor, Dion Hinchcliffe, Duane Nickull: *Web 2.0 Architectures: What entrepreneurs and information architects need to know.* O'Reilly Media / Adobe Dev Library. 2009

[5] Piero Fraternali, Sara Comai, Alessandro Bozzon, Giovanni Toffetti Carughi: *Engineering Rich Internet Applications with a Model-Driven Approach.*

[6] Charlie Robbins: *Scaling isomorphic javascript code.*
*http://blog.nodejitsu.com/scaling-isomorphic-javascript-code*

[7] Adam Welc Richard, L. Hudson, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai: *Generic Workers – Towards Unified Distributed and Parallel JavaScript Programming Model.* 2010

[8] Node.js *http://nodejs.org*

[9] Stefan Tilkov, Steve Vinoski: *Node.js: Using JavaScript to Build High-Performance Network Programs.* 2010

[10] Backbone.js *http://backbonejs.org*

[11] Dart *http://www.dartlang.org*

[12] TypeScript *http://www.typescriptlang.org*