POLITECNICO DI MILANO

Corso di Laurea Specialistica in Ingegneria Informatica

Dipartimento di Elettronica e Informazione

easyHTML5: Sviluppo di Applicazioni Web mediante MVVM

Relatore: Prof. Sam Jesus Guinea Montalvo

Tesi di Laurea Specialistica di:

Lorenzo Cavazzi

Matricola 754986

Anno Accademico 2011 – 2012

Milano, 4 ottobre 2012

Dedicata ai miei familiari per avermi sempre sostenuto e sollecitato nei momenti difficili.

Ringrazio il mio relatore, i professori, i compagni di studi e tutti gli amici veri che hanno contribuito in modi diversi al raggiungimento di questo importante traguardo.

Indice

Indice delle illustrazioni

ESTRATTO

Negli ultimi anni i pc hanno raggiunto un'ampia diffusione su scala mondiale, e sono stati affiancati da un numero sempre crescente di device elettronici dotati di elevata potenza computazionale. Questi dispositivi sono eterogenei fra loro, sia per quanto riguarda i sistemi operativi che li equipaggiano, sia per le modalità d'interazione che li caratterizzano. Creare software più o meno complessi con metodi tradizionali richiede progettazioni dedicate all'ambiente d'esecuzione per creare l'interfaccia utente e, in parte, anche per la logica di business.

In questo contesto si fa sentire l'esigenza di linguaggi in grado di essere interpretati su più sistemi operativi garantendo comportamenti uniformi, e uno dei candidati ad assolvere tale compito è HTML5. Esso non serve più solo per creare pagine web, ma l'ultimo aggiornamento (incluso quello alle tecnologie strettamente correlate, cioè JavaScript e CSS) permette di adottarlo per creare applicazioni arbitrariamente complesse, sia web based che desktop. Il vantaggio strategico che può vantare sui concorrenti è dovuto all'ampio supporto di cui gode tra i device di ultima generazione che, facendo dell'accesso a internet uno dei loro punti di forza, prevedono come requisito indispensabile la presenza di un browser.

Il problema a cui ci interessiamo riguarda la consistenza dei dati fra logica di business e interfaccia utente. Queste due parti sono solitamente progettate separatamente, secondo principi ormai assodati dall'ingegneria del software. Tuttavia nasce una dualità tra i contenuti degli elementi HTML e delle variabili JavaScript. I primi espongono all'utente lo stato delle seconde, ma l'eventuale

aggiornamento di una delle due componenti deve essere correttamente trasmesso a tutte le parti che potrebbero esserne influenzate. Nel caso di un elevato numero di dipendenze diventa difficile garantire la sincronia tra i dati coi quali sta interagendo l'utente e quelli al livello sottostante.

La nostra idea prevede l'utilizzo di un modello architetturale che ben si adatta agli ambienti event-driven (come HTML): il Model View ViewModel. Ci poniamo l'obiettivo di utilizzare un approccio diverso da quelli trovati in letteratura, cercando in particolare di sollevare il programmatore dal compito di dover scrivere tutti i controllori e determinare manualmente le dipendenze. Riassumendo in poche parole, vogliamo creare una libreria che, una volta realizzata la logica di business, sia in grado di generare automaticamente sia gli elementi dell'interfaccia che i suoi controllori, il tutto con l'aggiunta di poche righe di codice.

Sebbene uno strumento del genere rischia di essere poco flessibile, cerchiamo di garantire supporto a un elevato numero di tipologie di dato, e di sopperire ai limiti con con un'elevata dinamicità degli elementi dell'interfaccia che, oltre ad esporre il contenuto dei dati del modello sottostante, determinano le modalità d'interazione a disposizione dell'utente.

La libreria che abbiamo realizzato e che descriviamo in questo documento presenta ancora qualche lacuna, ma contiamo di proseguirne lo sviluppo in modo da renderla adatta alla creazione di applicazioni di reale utilità, al di fuori delle dimostrazioni in ambito accademico.

CAPITOLO 1

INTRODUZIONE

Nel corso degli ultimi anni i computer sono stati utilizzati in un numero crescente di campi applicativi, e ciò ha portato alla necessità di creare software in grado di soddisfare le esigenze più disparate. Il bacino d'utenza è quindi diventato sempre più ampio ed eterogeneo, con un cospicuo numero di persone non specializzate nell'uso del mezzo informatico.

Nasce quindi l'esigenza di rendere l'interfaccia utente, ovvero lo spazio d'interazione fra uomo e macchina, sempre più intuitiva sia per semplificare l'utilizzo del software che per rendere immediate operazioni una volta ben più complicate. Essa ha quindi subito un forte processo di evoluzione che negli anni l'ha portata da una semplice console per l'input/output dei comandi fino a un ambiente grafico in grado di guidare efficacemente l'utente verso il risultato voluto; non è più necessario conoscere tutti i comandi più o meno complessi, ma è sufficiente interagire con un'interfaccia grafica per stimolare la funzionalità desiderata [1].

Per ottenere questo risultato è necessario separare la progettazione di quest'ultima della logica applicativa, e questo disaccoppiamento è giunto a tal punto da rendere l'implementazione delle due parti dei processi quasi completamente separati [2]. Ciò comporta una serie di problemi, in particolare per quanto riguarda la consistenza dei dati. Si tenga presente che nelle

applicazioni le routine che modificano i dati possono essere lanciate sia dall'utente, tramite opportuni input, sia originate da cambiamenti di più basso livello (diverse condizioni dell'ambiente d'esecuzione, saturazione delle risorse hardware, o anche semplici modifiche alle variabili da parte di flussi di controllo). Le fonti che vanno a modificare gli stati possono essere molte per applicazioni complesse, e non è certo banale riuscire a controllarle tutte, specialmente nel caso di dipendenze a catena fra i dati.

1.1 Contesto

La consistenza dei dati è un problema che affligge tutti quegli ambiti dell'informatica dove la loro sorgente può esporre il contenuto a delle elaborazioni in un contesto differente da quello originale, col rischio di perdere la sincronia fra le due parti [3]. Il problema può diventare importante anche nella creazione di un'applicazione dove la separazione tra logica applicativa e interfaccia utente implica l'utilizzo di due linguaggi differenti. Tra i tanti esempi disponibili prendiamo quello delle applicazioni HTML. La scelta non è casuale ma bensì determinata dalla grande attenzione che sta ricevendo questo linguaggio grazie a una serie di punti a suo favore.

Per prima cosa precisiamo che quando si parla di programmazione HTML ci si riferisce all'insieme di tecnologie HTML, JavaScript e CSS, inizialmente pensate per le pagine web. Le specifiche dei 3 linguaggi sono state recentemente aggiornate in modo da poter supportare la creazione di applicazioni complesse sia online che locali (par 2.1.5). Inoltre la maggior parte dei nuovi dispositivi in commercio include già un browser e, in alcuni casi, è possibile eseguire solo applicazioni HTML5 anche in ambiente desktop [4][5].

La struttura del codice prevede che HTML esponga i contenuti tramite elementi con associati una serie di metadati, che CSS si occupi dell'impaginazione e della grafica, e che JavaScript sia il linguaggio di programmazione vero e proprio per creare la logica dell'applicazione (par 2.1).

Tuttavia viene a crearsi una dualità tra JavaScript e HTML, con il rischio di avere delle variabili che espongono a video il contenuto, ma una volta che l'utente interagisce con esse è possibile perdere la sincronia e maneggiare dati non più consistenti col livello sottostante. Diventa dunque necessario adottare un approccio sistematico per risolvere il problema, e uno strumento utile è rappresentato dai design pattern proposti dall'ingegneria del software [6].

In pratica essi rappresentano delle soluzione architetturali da adottare per risolvere problemi che si verificano di frequente (par 2.2.1, 2.2.2). Alcuni sono particolarmente adatti a mantenere la consistenza dei dati per applicazioni altamente interattive realizzate in un ambiente basato sugli eventi [7], per il quale JavaScript è particolarmente indicato.

1.2 Modalità d'intervento

In letteratura troviamo alcune proposte per porre rimedio ai problemi identificati sopra, in particolare ci sono delle librerie JavaScript che permettono d'implementare architetture MVC o MVVM mettendo a disposizione del programmatore delle API specifiche (par 2.2.3). Tuttavia quello che vogliamo proporre è uno strumento in grado di semplificare lo sviluppo in quegli ambiti dove sia facilmente automatizzabile buona parte della realizzazione dell'applicazione finale, in special modo per quanto riguarda sia i meccanismi di sincronismo tra i componenti (par 3.2.1), sia la creazione dell'interfaccia utente (par 3.2.2).

Ovviamente siamo consapevoli che un obiettivo del genere è piuttosto ambizioso, ma cerchiamo di puntare l'attenzione verso alcuni casi in cui l'approccio proposto garantirebbe più benefici che limitazioni. Per tipologie di dato standard (numeri, stringhe, date, boolean) spesso è facilmente prevedibile l'output che si vuole determinare basandosi solo su informazioni di contorno. Per esempio supponiamo di lavorare con un oggetto `studente` che prevede dei campi `nome`, `cognome`, `eta`, `facoltà`. E' probabile che, in base

al contesto d'esecuzione, si voglia dare la possibilità di modificare lo stato delle variabili ed eventualmente salvarlo permanentemente, oppure limitarsi a esporre gli attuali contenuti. In caso di abilitazione alle modifiche è prevedibile che, nel caso di variabili ancora vuote, si voglia creare campi di input (opportunamente adattati alla tipologia di dato da inserire) per ricevere nuove informazioni, mentre presumibilmente sarà più indicato un label (eventualmente modificabile) per visualizzare uno stato già definito. In questo scenario è possibile creare uno strumento in grado di valutare il contesto applicativo per generare automaticamente l'interfaccia utente a partire dal modello dei dati. Tale interfaccia, essendo generata senza l'intervento diretto del programmatore, presenta due grandi vantaggi: per prima cosa la gestione della sincronia dei dati è molto più facile; se gli elementi esposti sono creati direttamente da quelli del modello sottostante, è quasi immediato stabilire delle catene di dipendenze. Inoltre se l'elemento da esporre viene generato in base al contesto, è possibile modificare quest'ultimo o sostituirlo una volta a runtime nel caso di modifiche dinamiche a una qualsiasi variabile di contorno o allo stato stesso delle variabili.

Riassumendo vogliamo creare uno strumento in grado di mantenere la consistenza dei dati tra logica dell'applicazione e interfaccia utente, ma senza richiedere al programmatore di definire ogni volta dei controllori per le singole variabili, ma cercando piuttosto di generarli in automatico in tutti quei casi in cui un comportamento standard è sufficiente (par 3.1).

1.3 Implementazione e risultati attesi

Considerando l'ambito in cui ci muoviamo, ovvero la programmazione in HTML5, abbiamo deciso di creare una libreria JavaScript per estendere le funzioni di base del linguaggio con oggetti e metodi in grado di favorire un'immediata definizione dei componenti dell'architettura MVVM (par 4.1). In particolare lasciamo al programmatore il compito di creare il Model e le parti del ViewModel per le quali i comportamenti standard stabiliti dalla libreria non

sono adeguati (par 4.2.1), generando invece in automatico la View (par 4.2.3).

Uno degli obiettivi principali che ci poniamo si può identificare nella volontà di semplificare la creazione delle applicazioni e di ridurre al minimo il codice necessario per gestire gli elementi, per cui vengono forniti degli oggetti del Model per gestire anche contenuti multimediali come flussi video, audio, immagini e mappe interattive (par 4.1.3). Poichè molte applicazioni necessitano di sincronizzarsi con server in remoto, forniamo anche un oggetto in grado di gestire connessioni tramite l'elemento JavaScript WebSocket per supportare l'invio e la ricezione di contenuti da (e verso) un server (par 4.2.2).

Abbiamo deciso di utilizzare anche funzionalità attualmente ancora in approvazione presso il W3C [8] poiché queste sono già supportate da buona parte dei browser in circolazione (par 2.1.5), e sono fondamentali per poter implementare alcune funzionalità utili per creare applicazioni complesse o garantire la persistenza dei dati senza l'aiuto di un server (par 4.3.2).

Il lavoro svolto è supportato anche da una serie di strumenti dedicati all'apprendimento della sintassi introdotta nel linguaggio e a facilitare il testing delle funzionalità offerte da easyHTML5. Riteniamo molto importante questa parte e gli abbiamo dedicato una certa cura poiché speriamo che la libreria possa rilevarsi utile per creare applicazioni di utilità reale, non solo limitate all'ambito accademico per dimostrarne le potenzialità. Sono disponibili online esempi di confronto con altre soluzioni che adottano architetture MVVM (par 5.1.2), oltre a una documentazione esaustiva sia in pdf che in ScriptDoc (par 5.2.1), e dei tutorial interattivi che rappresentano un ottimo strumento per imparare a creare applicazioni con easyHTML (par 5.2.2).

Speriamo che il lavoro svolto sia sufficientemente completo, ma ci rendiamo conto che creare uno strumento realmente valido richiede molto lavoro e dei feedback da parte degli utilizzatori. L'intento è quello di proseguire nello sviluppo con nuove release che aumentino la flessibilità degli strumenti a disposizione. Inoltre sono in discussione dei progetti più ambiziosi: il primo riguarda la creazione di un ambiente di sviluppo dedicato alla composizione visuale della View, dalla quale derivare in seguito ViewModel e Model (l'approccio adottato sarebbe diametralmente opposto rispetto a quello attuale, dove si parte dal Model e poi si astrae); il secondo prevede la creazione di una controparte lato server per cercare di fornire un framework completo in

grado di supportare la persistenza dei dati lato server e fornire la possibilità di far accedere contemporaneamente più utenti allo stesso Model remoto, garantendo il mantenimento del sincronismo verso tutte le View interattive.

CAPITOLO 2

STATO DELL'ARTE

La diffusione del mezzo informatico in ogni campo applicativo che coinvolga l'elaborazione o lo scambio d'informazione ha portato all'allargamento del bacino d'utenza che si trova a interagire con il pc o altri device elettronici. Si è così venuta a creare creare la necessità di proporre interfacce semplici, immediate e intuitive per tipologie d'utente anche molto differenti fra loro [9]. La difficoltà di questo compito diventa ancora più grande se si pensa alla sempre crescente complessità delle applicazioni, che cercano di andare a coprire ogni necessità elaborativa. Inoltre sviluppare codice per dispositivi eterogenei richiede un lavoro extra per garantire la compatibilità in sistemi operativi differenti, e la necessità di adattarsi allo schermo del device che verrà utilizzato per l'output e per le interazioni. In questo contesto la progettazione dell'interfaccia utente diventa quasi disaccoppiata dalla logica del programma, in modo da non limitare le potenzialità del software. Dei benefici di un tale approccio si occupa l'ingegneria del software, e in letteratura troviamo diverse discussioni a riguardo [10].

Il lavoro presentato in questo documento riguarda un contesto che sta vivendo una forte espansione, ovvero quello delle applicazioni all'interno dei

browser (sia web based che pensate per un ambiente locale). Tale crescita è dovuta in buona parte alla possibilità di eseguire il codice sulla quasi totalità delle piattaforme introdotte di recente in commercio che, sfruttando internet e le connessioni veloci, fanno del browser un elemento indispensabile. A fianco a questo fattore di successo ne troviamo un secondo altrettanto fondamentale, ovvero il rinnovo degli standard introdotti per il web, inclusi quelli che, al momento della stesura di questo testo, sono ancora allo stato di draft ma prossimi all'approvazione.

Tale evoluzione però sta evidenziando una carenza di strumenti dedicati alla realizzazione di interfacce che garantiscano la consistenza dei dati con il modello sottostante. Considerando il continuo allargamento del bacino d'utenza, è facile intuire come questa lacuna per gli sviluppatori possa rappresentare un forte limite. Cerchiamo ora di presentare meglio il contesto in cui andiamo a operare e i limiti attuali nello stato dell'arte, per poi passare alla soluzione proposta e all'implementazione.

2.1 PROGRAMMAZIONE IN HTML

Il termine HTML (HyperText Markup Language) si riferisce nello specifico al meta-linguaggio usato per i documenti ipertestuali disponibili nel World Wide Web. In realtà quando si parla di programmazione in HTML ci si riferisce generalmente alla creazione di programmi che si basano sulle tecnologie che vanno a costituire le pagine realizzate per i browser, ovvero HTML, CSS e JavaScript. I 3 linguaggi sono stati introdotti come standard per le pagine web, e coesistono fra di loro con compiti differenti. HTML espone il contenuto, avvalendosi di etichette chiamate tag per identificare la tipologie di dato e salvare metadati aggiuntivi; CSS (Cascading Style Sheets) è una sintassi che permette di creare regole per l'impaginazione, o più in generale serve a personalizzare in ogni dettaglio la disposizione a video e la grafica degli elementi; JavaScript è il linguaggio di programmazione vero e proprio, che permette di implementare tutta la logica applicativa e quindi i contenuti più avanzati.

Alla loro introduzione i browser servivano solo come strumento per accedere a documenti presenti su internet, i quali erano formati per lo più da testo, tabelle e immagini [11]. Il supporto era quindi limitato alla fruizione di contenuti, con scarse possibilità d'interazione. Solo successivamente, con l'evoluzione tecnologica e la diffusione di internet su larga scala [12], il browser è diventato lo strumenti di accesso per contenuti più avanzati, tra cui file multimediali e applicazioni sempre più complesse. Per gestire queste novità si è reso necessario adattare gli standard, che sono rimasti fermi per troppo tempo (basti pensare che le specifiche HTML risalgono al 1999, mentre quelle CSS all'anno successivo), cosa che ha determinato la nascita di strumenti non standard per sopperire alle mancanze, col risultato di rendere le pagine web pesanti e scarsamente compatibili [13].

Con l'introduzione di HTML5, CSS3 ed EcmaScript 5, i browser moderni hanno ora le potenzialità per eseguire applicazioni potenti e versatili, col vantaggio di essere in una posizione privilegiata sia per la programmazione di applicazioni web oriented, sia per la compatibilità tra diversi sistemi operativi.

2.1.1 HTML5

Vediamo ora più nel dettaglio le novità introdotte dal nuovo standard HTML5. Si fa presente che le specifiche ufficialmente approvate dal W3C, l'organo predisposto alla definizione degli standard per il World Wide Web, sono ancora ferme ad HTML4, mentre quelle di cui discutiamo ora sono ufficialmente allo stadio di working draft, e l'approvazione definitiva è prevista entro il 2014 [14]. Tuttavia le ultime versioni dei browser offrono già un ottimo supporto a tutte le tecnologie giunte allo stadio di Candidate Reccomandation o Last Call (par 2.1.5).

Per prima cosa HTML5 introduce pieno supporto agli elementi multimediali con i tag audio e video, così da eliminare le soluzioni basate su linguaggi proprietari che sono state create in tempi recenti per colmare questa lacuna. I visualizzatori di video in Flash o in Silverlight presentano una serie di problemi: per prima cosa necessitano l'installazione dell'interprete per il browser sulla macchina locale, limitando la compatibilità e aumentando molto la pesantezza [15]. Inoltre, essendo implementazioni non standard, spesso presentano comportamenti eterogenei fra di loro, generando confusione nell'utente.

Vengono introdotti nuovi elementi anche per la gestione delle immagini, fornendo il supporto per la grafica vettoriale SVG e per la creazione di elementi avanzati tramite JavaScript. Nel primo caso si vuole garantire un rendering di alto livello a prescindere dalle dimensioni dell'output, in modo da evitare immagini distorte come spesso accade utilizzando i formati tradizionali. Si pensi ad esempio a un logo in JPEG che, una volta ingrandito il documento, risulta sgranato e poco professionale. Le immagini programmabili tramite JavaScript sono invece necessarie per creare applicazioni altamente complesse da un punto di vista grafico, come per esempio i videogiochi. L'elemento `canvas` permette anche l'accesso alle risorse hardware di basso livello così da consentire l'accelerazione grafica tramite scheda video [16].

Per la programmazione di applicazioni che necessitano il salvataggio in locale di una certa quantità di risorse viene introdotto il web storage, che consente di salvare delle coppie chiave valore. E' possibile mantenere i dati in modo permanente o limitarsi alla durata della sessione, o ancora salvare dati

per una specifica istanza dell'applicazione, consentendo di eseguirne più di una contemporaneamente all'interno dello stesso browser. Con i file manifest si può anche salvare interamente l'applicazione in locale, così da poter essere utilizzata come un normale programma desktop. In caso di task particolarmente pesanti sono resi disponibili gli web workers, che implementano funzionalità simili a quelle dei thread, mentre per garantire l'interazione con il resto dell'ambiente si può ricorrere ai nuovi eventi di Drag&Drop [8].

La gestione dei contenuti da remoto è stata notevolmente migliorata, introducendo sia novità per le chiamate AJAX (si passa alla versione 2 dell'elemento XMLHttpRequest) che un elemento del tutto nuovo, il WebSocket. Questo consente la comunicazione bidirezionale con un server, instaurando un canale full duplex che utilizza una modalità di scambio dati simile a quella dell'HTTP, consentendo il push di dati dal server senza ricorrere a COMET o altri metodi che necessitano la stimolazione di nuovi messaggi da parte del client [17].

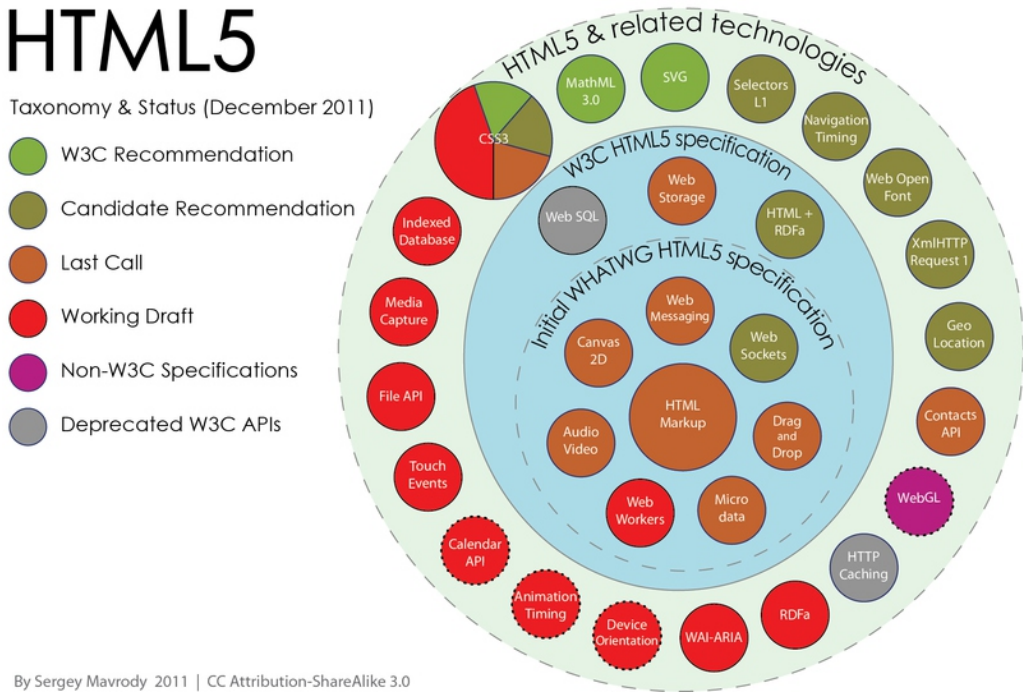

Figura 2.1: stato di avanzamento nell'approvazione da parte del W3C delle tecnologie relative ad HTML5

Infine vengono introdotti una serie di nuove elementi e di API per gestire tipologie di dato sempre più frequenti in internet. Tra questi troviamo il supporto alla geolocalizzazione, all'impaginazione di stampo giornalistico, elementi per uniformare la gestione di date, password, contenuti cifrati, e una serie di altre tipologie di dato. Ora è possibile anche la suddivisione semantica dei contenuti utilizzando i tag, e non più basandosi sugli id, in modo da ottenere un maggiore livello di standardizzazione in tutto il web. In figura 2.1 ricapitoliamo gli standard proposti e il loro stato di avanzamento nell'approvazione finale.

2.1.2 JavaScript ed ECMAScript 5

JavaScript è il linguaggio di scripting nato per affiancare l'HTML nelle pagine web, adatto ad intervenire direttamente sugli elementi del DOM (Document Object Model, ovvero il modello del documento che contiene l'albero degli elementi HTML) e a gestire la programmazione basata su eventi [18]. La sua caratteristica principale è quella di essere un linguaggio interpretato, ovvero il codice non viene compilato ma interpretato a runtime. Questo permette di scrivere codice in grado di compiere sul browser operazioni complesse e di poter interagire con l'ambiente locale nel quale viene eseguito. Un altro vantaggio, utile nel caso nel caso di applicazioni web based, è quello di poter sgravare il server da parte del lavoro.

La sintassi è relativamente simile a quella degli altri linguaggi di programmazione C-like come il Java ma, nonostante la somiglianza nel nome con quest'ultimo, i 2 sono del tutto indipendenti e tra di loro ci sono comunque forti differenze. Per prima cosa il JavaScript è un linguaggio debolmente tipizzato e debolmente orientato agli oggetti [19]. Gli oggetti infatti sono più simili agli array associativi del Perl, mentre il concetto di classe non esiste, o meglio non è inteso nel senso classico, rendendo così le strutture dati molto più flessibili. Di conseguenza anche l'ereditarietà è differente, e si riduce a una copia dei metodi dall'oggetto padre a quello figlio [20].

JavaScript è basato sulle specifiche dello standard ufficiale ECMAScript dal quale derivano altri dialetti come JScript ed ActionScript. ECMAScript è stato aggiornato di recente alla versione 5 dopo anni di scarse innovazioni, ed è ora molto più vicino alle capacità degli altri linguaggi object oriented più diffusi, pur mantenendo una flessibilità decisamente maggiore [21]. Attualmente è utilizzato sempre più spesso al di fuori dei browser: un numero rilevante di software consente infatti di creare script in JavaScript per l'estensione delle funzionalità [22], mentre alcuni sistemi operativi emergenti consentono la creazioni di applicazioni con questo linguaggio (par 2.1.5), anche nel caso in cui debbano operare completamente offline.

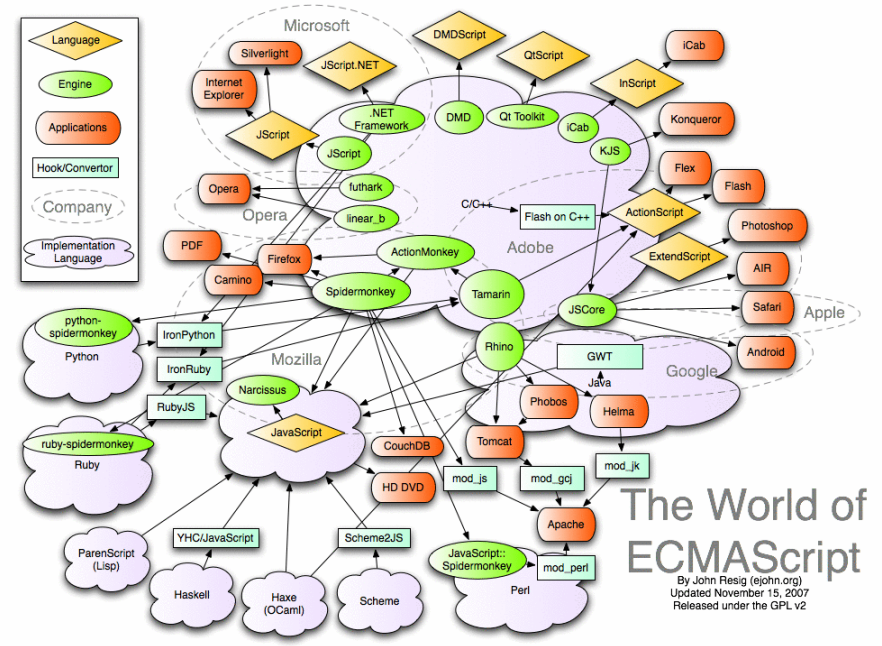

Figura 2.2: ECMAScript e i suoi dialetti

JavaScript è un linguaggio general purpose, e con esso è possibile creare ogni genere di applicazione. Generalmente il codice viene inserito nei documenti HTML all'interno di appositi tag `script`, ma si possono anche creare file contenti solo JavaScript (con estensione `.js`) ed importarli nei documenti. Per estendere le funzionalità immediatamente disponibili è possibile inserire nelle pagine delle librerie o dei framework arbitrariamente complessi. Tale approccio risulta molto comodo, poiché consente di estendere il linguaggio che ha serie di costrutti di base abbastanza limitato. Non è necessario aggiungere le librerie in posizioni particolari (come avviene per altri

linguaggi Object Oriented), poiché l'interprete andrà a leggere ed eseguire ogni volta tutto il codice (incluso quello delle fonti esterne), permettendo così di aggiungere solo le funzionalità che interessano per lo specifico task che dovrà essere eseguito all'interno di una pagina.

Ci sono alcune librerie che sono diventate molto popolari in determinati ambiti poiché le API che mettono a disposizione sono comode e al contempo semplici da utilizzare. Un esempio su tutti è jQuery, che implementa funzionalità dedicate alla manipolazione degli elementi del DOM in modo decisamente più semplice e immediato rispetto a quanto avviene con le tradizionali funzioni JavaScript. Per questo motivo può vantare una diffusione che ha di recente superato il 50% del totale di tutte le pagine web [23], un risultato impressionante che non ha eguali in soluzioni concorrenti. Tuttavia ci sono molte altre librerie che, pur godendo di uno share decisamente inferiore, nel loro campo sono ampiamente utilizzate e conosciute, poiché riescono ad andare incontro a esigenze molto specifiche. Tra di esse troviamo esempi come JavaScript MVC o BackboneJS, che implementano dei framework per l'utilizzo del design pattern MVC, oppure Knockout, che invece si basa sull'architettura MVVM, più recente della precedente e che sta riscuotendo un grande successi nelle applicazioni web (par 2.2.3).

2.1.3 CSS 3

Un'altra tecnologia già introdotta precedentemente (par 2.1) che va ad affiancare HTML e JavaScript nello sviluppo delle applicazioni web per browser sono i fogli di stile (Cascading Style Sheets), che vengono utilizzati per definire la formattazione di documenti HTML e XML. L'introduzione delle specifiche CSS si è resa necessaria per separare i contenuti dalla formattazione, così da superare i limiti di controllo sulla grafica inizialmente presenti in HTML [24].

Anche in questo caso la versione precedente dello standard era ferma da quasi 10 anni, lasciando una profonda lacuna per quanto riguarda gli elementi

animati, quelli in 3 dimensioni o altre tipologie altamente dinamiche che sono diventate ormai comuni e talvolta indispensabili per ottenere effetti efficaci sia da un punto di vista visivo che per quanto riguarda l'usabilità [25]. Vediamo le principali innovazione già introdotte o in fase di definizione.

Per prima cosa sono state aggiunte molte proprietà su cui intervenire per quanto riguarda gli elementi testuali. Ad esempio è possibile inclinare i caratteri, deformarli su ognuna delle 3 dimensioni, applicare ombre o distorsioni specifiche. Molti di questi effetti erano disponibili solo tramite codice JavaScript (tra l'altro ottenibili solitamente in modo tutt'altro che banale), finendo per obbligare il programmatore a mischiare la grafica alla logica del programma. O ancora peggio si cercava di risolvere il problema creando immagini che rappresentano del testo, col grave difetto di avere una grandezza fissa (quindi scarsa scalabilità dimensionale), occupare molto spazio (in termini di KB) e non essere interpretabili come elemento testuale dal browser (quindi rendendo disponibili interazioni poco appropriate).

Per adeguarsi al numero crescente di blog e testate online, sono in fase di definizione anche delle proprietà per gestire l'impaginazione di stampo giornalistico. Il testo può essere suddiviso in più colonne in modo automatico, per adattarsi a dispositivi di output eterogenei, e i font non devono più risiedere in locale, così da azzerare le differenze tra i diversi contesti d'esecuzione. I contenuti diventano deformabili sia in modo statico (contenitori allungati, bordi arrotondati, sfondi differenti, ...) che tramite animazioni, ed è possibile creare comportamenti che si ripetono nel tempo, simulando tutte le distorsioni che in precedenza dovevano essere create con JavaScript sfruttando gli eventi del DOM. Anche i colori sono interessati da queste modifiche, e tra i loro parametri compare ora la trasparenza, in modo da rendere semplici effetti come la dissolvenza o il fade-in/fade-out.

Per finire uno dei cambiamenti più importanti riguarda il posizionamento degli oggetti nell'interfaccia, che in precedenza rispettavano l'ordine del documento HTML. Gli attributi `float` e `position` consentono da sempre di far scorrere gli elementi a destra o sinistra, oppure di fissarli sulla pagina, ma non di invertire la disposizione di oggetti inseriti dopo nel DOM. Ora diventa possibile gestire in modo completamente customizzato la disposizione a video, eventualmente cambiandola in base al device di output.

2.1.4 Applicazioni complesse

Dopo aver dato una descrizione generale delle potenzialità dei nuovi strumenti, vogliamo ora portare l'attenzione sulle applicazioni più o meno complesse che è possibile realizzare. Cerchiamo di spiegare quale sia l'approccio migliore per la creazione di programmi che prevedono la possibilità di ricevere modifiche ai dati utilizzati sia dall'interfaccia utente che dalla logica applicativa.

Dai paragrafi precedenti si deduce che JavaScript è un linguaggio general purpose, dunque adatto a realizzare quasi tutte le tipologie di applicazione. Nonostante la flessibilità dei nuovi strumenti resi disponibili dall'aggiornamento degli standard, esso viene utilizzato abitualmente non solo per la logica applicativa, ma anche per gestire il corretto posizionamento degli elementi nell'interfaccia ed eventuali effetti grafici. Questo viene fatto sia per abitudine, poiché in passato non era possibile fare altrimenti, sia per motivi di retrocompatibilità coi vecchi browser, che sono ancora molto diffusi (fig 2.3) e non supportano i nuovi elementi e le ultime regole CSS in modo nativo.

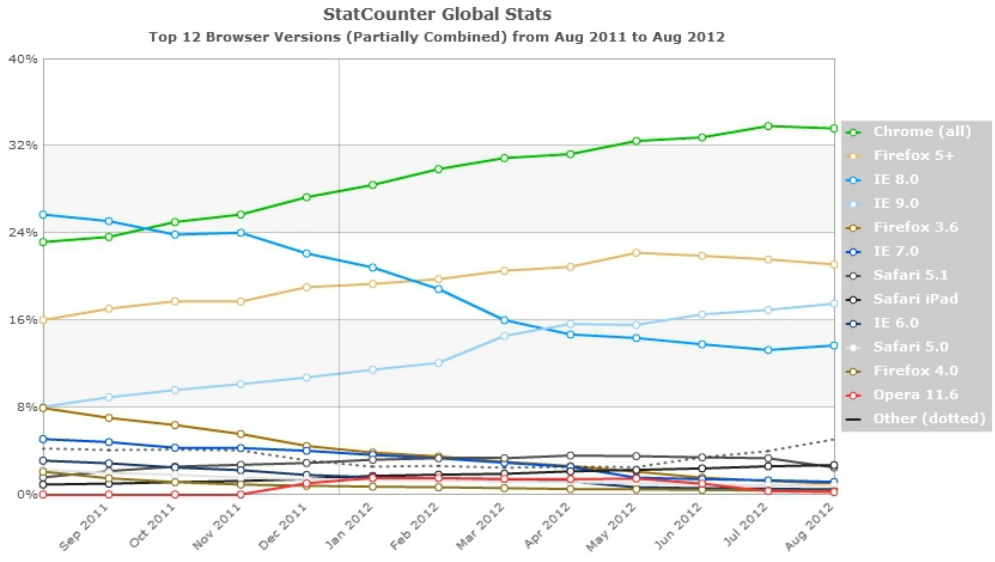

Figura 2.3: market share dei browser, suddivisi per versione

Tuttavia questo approccio rende confusionario lo sviluppo dell'applicazione e, invece di separare logica e interfaccia, si finisce col gestire in modo disordinato l'integrità dei dati tra elementi diversi, o peggio ancora non viene

garantita la consistenza tra contenuto HTML e variabili JavaScript, che va persa al primo cambiamento da parte di una delle due fonti.

In questo contesto sarebbe opportuno estendere l'adozione dei design pattern concepiti dai teorici dell'ingegneria del software [10]. Tale approccio si rende necessario alla luce della crescente complessità delle applicazioni browser based che, non essendo banali come quelle di qualche anno fa, richiedono spesso lo sviluppo all'interno di un team formato da più persone, e hanno un ciclo di vita che tende a diventare sempre più lungo (e quindi richiede aggiornamenti sistematici). Approfondiamo questo punto nel paragrafo 2.2, ma prima vale la pena contestualizzare meglio gli ambiti di utilizzo delle applicazioni HTML5.

2.1.5 Ambiti di utilizzo e compatibilità

HTML5, come già sottolineato, rappresenta l'ultima evoluzione degli standard per la creazione delle applicazioni browser based, con l'ambizioso scopo di diventare un punto di riferimento per lo sviluppo di applicazioni multipiattaforma. Non è un caso che l'evoluzione stia avvenendo proprio in questo periodo in cui nuove tipologie di device (come tablet e smartphone) stanno conoscendo un'enorme crescita [26], rendendo disponibile grande potenza di calcolo su piattaforme del tutto nuove dotate di sistemi operativi eterogenei (fig 2.4). Normalmente questa diversificazione crea una barriera nella distribuzione del software sviluppato con tecniche tradizionali, che richiede una compilazione ad hoc in base al sistema operativo nel quale dovrà essere utilizzato [27]. JavaScript, essendo un linguaggio interpretato, non soffre di questo handicap, e può essere eseguito praticamente su ogni dispositivo consumer introdotto di recente sul mercato [28][29].

Questo vantaggio ha convinto i colossi dell'informatica a puntare fortemente sul supporto alle tecnologie HTML5, così da abbattere il problema della scarsa diffusione di applicazioni che può verificarsi quando vengono creati nuovi sistemi operativi che presentano forti differenze da quelli già

esistenti.

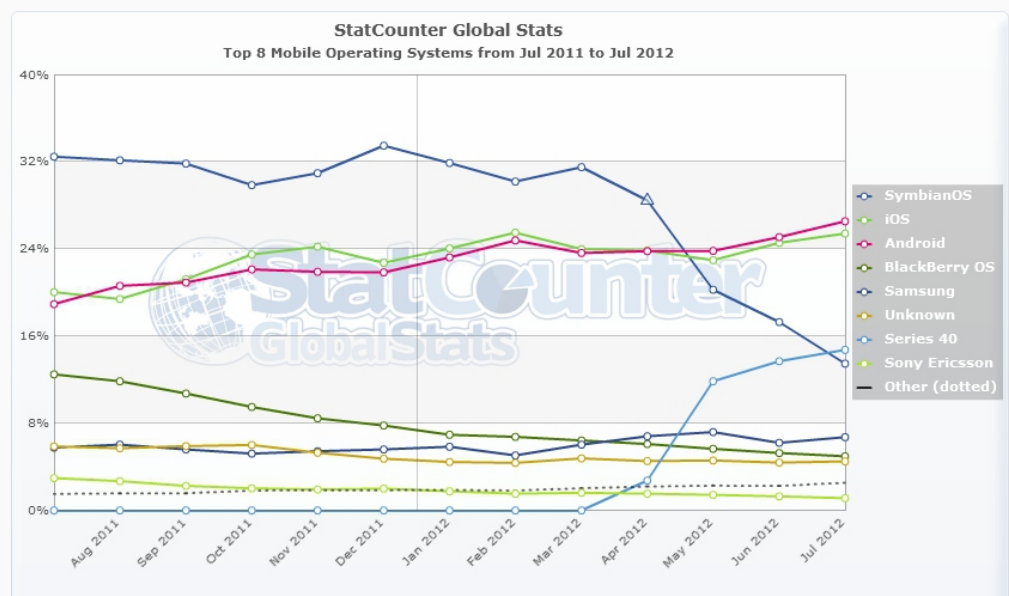

Figura 2.4: distribuzione eterogenea dei sistemi operativi mobile

Citiamo ad esempio Microsoft che lancerà a giorni il suo nuovo sistema operativo Windows 8 con un'interfaccia rinnovata (Metro interface) dedicata principalmente a sistemi touchscreen. La creazione delle applicazioni Metro è possibile con i linguaggi .NET ma anche, e in via preferenziale, con HTML5 [30]. In ambito mobile invece c'è grande attesa per il nuovo sistema operativo Tiezen, attualmente disponibile solo in versione beta, che dovrebbe porsi come concorrente di Android e iOs, spinto da colossi dell'IT come Intel, Samsung e Nokia. Esso prevede la possibilità di creare e compilare codice solo in HTML5, ponendolo di fatto come unico linguaggio di programmazione disponibile per la piattaforma, e quindi da utilizzare anche per tutte le applicazioni offline [4].

La possibilità di creare software con la garanzia di un'ampia compatibilità del codice è un enorme passo avanti sia per i programmatori che per l'utente. I primi possono creare un unico codice senza preoccuparsi di dover gestire separatamente ambienti differenti, e vedono allargarsi il bacino di potenziali clienti; i secondi hanno la garanzia di aver accesso a tutti i prodotti disponibili senza preoccuparsi dei limiti imposti dalle logiche di mercato.

Vale la pena spendere qualche parola anche sulla compatibilità delle applicazioni, che rappresenta un punto dolente in queste prime fasi di vita di

HTML5. Per prima cosa bisogna considerare il supporto da parte dei browser che rappresentano, almeno per ora, l'unico strumento in grado di eseguire applicazioni HTML. Come si vede in figura 2.5, anche le ultime versioni rilasciate dagli sviluppatori non hanno pieno supporto per molti elementi definiti dal W3C. Del resto, considerando che lo standard non è ancora definitivo e che sta subendo ancora un certo numero di cambiamenti [8], diventa difficile garantire il pieno supporto a tutte le novità.

Figura 2.5: supporto dei browser più diffusi alle specifiche HTML5

Considerando che le specifiche possono essere interpretate in modo più o meno ambiguo, le soluzioni proposte non sono sempre omogenee. Ad esempio l'attributo `contenteditable`, che dovrebbe poter rendere modificabile ogni elemento HTML, non stabilisce le esatte modalità d'interazione da mettere a disposizione dell'utente. Come conseguenza browser differenti come Chrome e Opera si comportano in modo simile in caso di testi editabili, ma ben differente in caso d'immagini. Di queste diversità bisogna tener conto, anche se col passare del tempo e con la definitiva approvazione dello standard HTML5, tali problemi dovrebbero sparire.

2.2 La consistenza dei dati

La consistenza è un problema che riguarda quei dati che vengono utilizzati contemporaneamente da diverse fonti. Evitare la perdita del sincronismo è un compito più o meno arduo in base alla struttura del software che si vuole creare, alle modalità d'interazione possibili, e in parte anche al linguaggio di programmazione utilizzato [3]. Nel caso di HTML5 il problema è particolarmente sentito poiché il documento HTML espone i contenuti, ma le operazioni complesse vengono eseguite tramite codice JavaScript. Sebbene quest'ultimo possa intervenire direttamente sugli elementi HTML, solitamente i dati vengono salvati in variabili locali, così da evitare problemi di conversione e soprattutto per non intervenire ogni volta sul DOM. In quest'ultimo caso le operazioni risultano essere intense dal punto di vista computazionale, quindi andrebbero evitate per quanto possibile [31]. Viene quindi a crearsi in modo naturale una dualità fra i dati visualizzati nell'interfaccia e quelli utilizzati dalla logica dell'applicazione, e il problema diventa grave quando lo stato delle variabili può essere modificato sia dall'utente, che interviene sull'interfaccia del browser, sia da cambiamenti al livello applicativo, magari determinati da fonti remote con cui si vuole restare sincronizzati.

Per gestire correttamente la consistenza è necessario creare del codice che si occupi in modo specifico di tale compito. Questo non dovrebbe interferire con la logica dell'applicazione, e consentire di mantenere un'interfaccia al contempo intuitiva e sempre aggiornata. Per soddisfare i requisiti appena citati diventa quasi indispensabile un approccio che si avvale di un buon modello architetturale. L'uso di design pattern, molto diffuso nella programmazione classica ma ancora poco adottato in HTML5, consente di ottenere applicazioni robuste per quanto riguarda la consistenza e l'integrità dei dati [6].

2.2.1 Model View Controller

Nel paragrafo precedente abbiamo accennato ai design pattern senza però esplicitare cosa siano. Con questo termine si indica un modello da applicare per risolvere un problema non banale che può presentarsi in diverse situazioni durante la progettazione e lo sviluppo del software [32]. In particolare siamo interessati ai pattern architetturali, che prevedono la creazione di schemi di base per impostare l'intera struttura di un software. Solitamente questi schemi corrispondono a dei componenti che ricoprono un ruolo specifico. Alcuni design architetturali sono ben consolidati (si pensi all'architettura client-server, o a quella basata sui layers), altri sono stati introdotti più di recente per rispondere a nuove esigenze nello sviluppo di software (i repository stanno riscuotendo grande successo grazie alla diffusione di Internet). Nello specifico ci interessano i pattern dedicati alla creazione di software con interfaccia utente e logica applicativa separate.

Il Model View Controller (comunemente abbreviato in MVC) è un pattern che prevede la presenza di 3 componenti separati fra loro ma in grado d'interagire per mantenere la consistenza dei dati. Il **Model** si occupa dell'accesso ai dati dell'applicazione, determinandone le strutture e salvando gli stati delle variabili. Più in generale possiamo dire che esso implementa la logica dell'applicazione (anche chiamata logica di business). La **View** è l'output, ovvero la visualizzazione del Model esposta all'utente in modo possibilmente efficace (ad esempio dei dati di vendita potrebbero essere esposti efficacemente come un grafico del volume prodotto sul periodo di tempo). A uno stesso Model potrebbe corrispondere più di una View tra le quali scegliere in base a qualche fattore discriminante: ad esempio potrebbe dipendere dalla grandezza dello schermo del device, oppure dalla risorse del dispositivo di output, o ancora dalle restrizioni legate alla tipologia di utente che deve interagire. Infine il **Controller** si occupa di recepire gli input dell'utente e convertirli in comandi per modificare Model o View.

Con l'aiuto dello schema in figura 2.6 cerchiamo di spiegare meglio le interazioni che avvengono fra le 3 componenti. La View è un elemento passivo, che si aggiorna in caso di segnalazioni da parte di qualche fonte. Il Model invece è generalmente attivo, e segnala i cambiamenti alla View in

modo che possa aggiornare l'output, e al Controller per cambiare eventualmente i comandi a disposizione dell'utente. Eventualmente il Model può essere passivo in caso non siano previsti cambiamenti che provengono direttamente dalla logica dell'applicazione. Il Controller viene invece pilotato dall'utente, e può andare ad alterare i dati del Model (che potrebbero poi essere propagati) o quelli della View (che potrebbe a sua volta modificare il modo in cui presenta i dati, ad esempio scorrendo in avanti l'insieme di dati dell'Array associato nel Model).

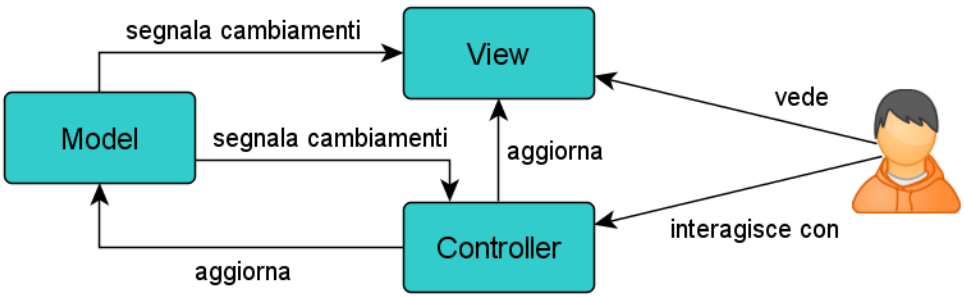

Figura 2.6: struttura dell'architettura MVC

Il MVC è stato originariamente impiegato dal linguaggio Smalltalk negli anni '80, per poi essere sposato da numerose tecnologie [33]. Nell'ambito delle applicazioni HTML il modello MVC ha ottenuto un discreto successo, e sono disponibili dei framework che implementano la logica appena vista. La maggior parte di essi sono dedicati alle web application e forniscono delle funzioni, all'interno di librerie JavaScript, per sincronizzare i dati col server, (dove si presuppone risieda il Model, tutto o in gran parte) e gestire sul client la creazione del Controller e della View. Citiamo le 2 più popolari: JavaScriptMVC e Backbone. Entrambe sono debolmente basate sul pattern MVC e implementano un'interfaccia RESTful per lo scambio di dati in formato JSON [34][35].

```
// Esempio di codice in BackboneJS

// definizione del Model
var GreetingModel = Backbone.Model.extend({
      defaults: {
            text: 'hello world'
      }
});
```

```
// definizione della View ed esposizione
var GreetingView = Backbone.View.extend({
	render: function() {
		$('p').html(this.model.get('text'));
		return this;
	}
});
var greet = new GreetingModel();
new GreetingView({model: greet}).render();

// definizione del controller
var RecordView = Backbone.View.extend({
	tagName: 'tr',
	events: {
		'click': 'performed';
	},
	initialize : function() {
		this.model.bind('change', this.render, this);
	},
	render: function() {
		// [...]
		return this;
	},
	performed: function() {
		this.model.set('done', true);
	}
});
```

In questo esempio di codice JavaScript si fa uso della libreria BackboneJS per creare un'applicazione “hello world” con un elemento del Model, uno della View e una parte del Controller. Senza entrare nei dettagli si nota che la sintassi è molto prolissa; sebbene ciò consenta un controllo altamente personalizzato su ogni aspetto dei componenti, la realizzazione di applicazioni richiede maggior tempo per la stesura del codice, e anche un periodo di apprendimento per imparare ad utilizzare tutte le funzionalità messe a disposizione. Considerando che il costo dello sviluppo ha un'importanza notevole, e che questo dipende direttamente dal tempo necessario, l'utilizzo del pattern MVC potrebbe non essere la scelta migliore ne caso in cui si vogliano creare delle applicazioni altamente interattive ma con comportamenti standardizzabili (ovvero dai comportamenti facilmente prevedibili per elementi simili). Si pensi ad esempio a un calendario per la selezione delle date, o a un'input per inserire l'età. La creazione di questi elementi dovrebbe essere immediata, e non richiedere molte righe di codice peraltro poco intuitive.

2.2.2 Model View ViewModel

L'architettura Model View ViewModel (abbreviato in MVVM) è stata introdotta nel 2006 da Microsoft per piattaforme che prevedono una programmazione basata su eventi (come appunto HTML5), e in particolare per lo sviluppo di software che richiede interfacce utente altamente interattive [36]. MVVM deriva direttamente da MVC del quale mantiene 2 componenti con compiti quasi invariati, mentre il terzo, ovvero il ViewModel, non è altro che un Controller specializzato nella gestione dello scambio dati e del sincronismo tra gli altri 2 elementi.

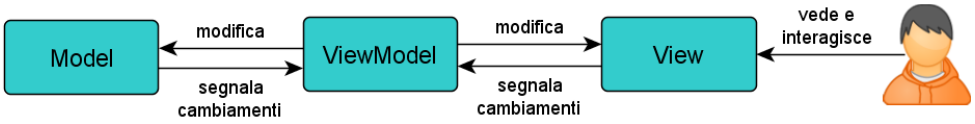

Figura 2.7: struttura dell'architettura MVVM

Aiutiamoci anche in questo caso con un semplice schema che spiega le interazioni. Come si vede in figura 2.7 il Model e la View non sono più in diretta comunicazione tra loro come avveniva per l'architettura MVC (fig 2.6). Il **Model** resta il responsabile della struttura dati e solitamente anche dello stato delle variabile, ma parte della logica di business non è più al suo interno poiché non deve modificare direttamente gli altri componenti, ma limitarsi a trasmettere delle notifiche. La **View** diventa un componente più attivo poiché, oltre a esporre l'output, si occupa di catturare gli eventi dell'interfaccia utente, anche se non li gestisce direttamente ma si limita a segnalarli al livello sottostante. Il **ViewModel** fa da tramite fra gli altri 2 componenti, occupandosi di propagare i cambiamenti e mantenere il sincronismo fra i layer. Di fatto incorpora parte della logica di business, poiché deve intervenire sulla View cambiando eventualmente le interazioni possibili e i valori esposti, e sul Model per applicare eventuali aggiornamenti allo stato delle variabili.

Il ViewModel, teoricamente, è un elemento più vicino alla View che al Model, poiché è una sua astrazione (come dice il nome dovrebbe essere una specie di modello di ciò che è contenuto nella View). C'è quindi una correlazione più stretta tra gli elementi della View e del ViewModel che tra

questi ultimi e quelli del Model. Gli elementi dei diversi componenti sono legati tra loro tramite binding, ovvero relazioni bidirezionali che vincolano lo stato di uno a quello dell'altro. Il concetto di binding è fondamentale, poiché stabilisce delle dipendenze che permettono al ViewModel di sapere quali relazioni ci sono tra elementi che, negli altri componenti, appaiono come distinti. In pratica permette di associare ogni modifica a degli eventi di update per tutte quelle quelle parti che ne sono potenzialmente influenzate [7].

2.2.3 Knockout

Nel paragrafo precedente abbiamo presentato l'architettura MVVM da un punto di vista puramente teorico. I componenti, così come sono stati descritti, dovrebbero svolgere dei compiti distinti, e il ViewModel dovrebbe essere l'unico elemento che effettua cambiamenti anche negli altri layer, mentre Model e View dovrebbero limitarsi a notificare delle variazioni nello stato. Inoltre ci si aspetta che la maggior parte dei cambiamenti vengano dalla View, e quindi effettuati dall'utente. Nelle implementazioni di questo pattern in HTML5 le cose vanno però in modo un po' differente.

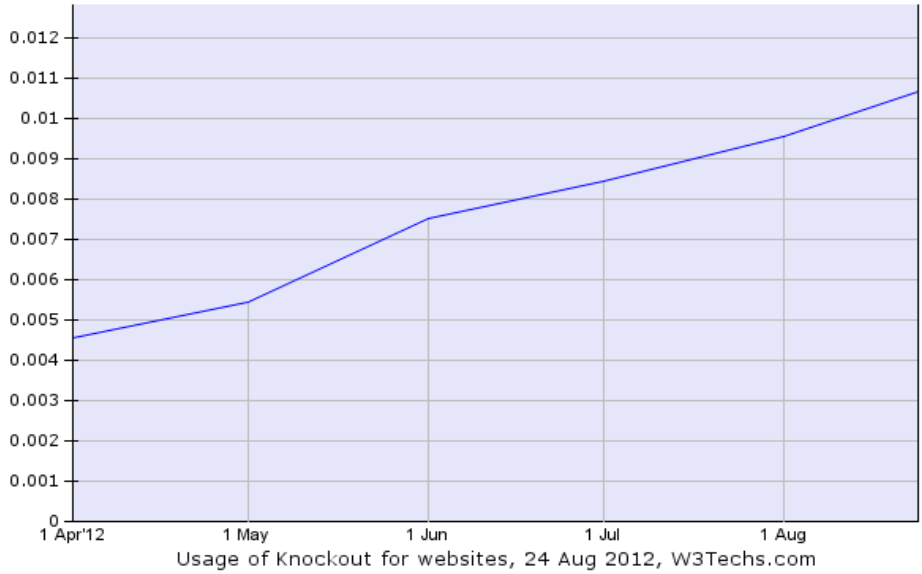

Figura 2.8: l'utilizzo di KnockoutJS è in fortissima crescita

Prendiamo come riferimento una libreria JavaScript MVVM piuttosto recente ma che sta riscuotendo un grande successo (fig 2.8), ovvero Knockout. Essa rappresenta lo stato dell'arte per creare applicazioni HTML5 adottando formalmente l'architettura MVVM [37], e di fatto anche l'unica soluzione del suo genere che si può definire sufficientemente matura da essere impiegata in progetti di una certa importanza [38].

In Knockout la creazione del Model e del ViewModel non è nettamente separata. Formalmente viene creata una funzione che contiene tutta la logica applicativa, che dovrebbe definire la modalità di gestione degli elementi del ViewModel (comportamenti dell'interfaccia, relazioni fra elementi, ...). Di fatto però si finisce con creare al suo interno anche la struttura degli elementi del Model e a definire il loro stato iniziale, senza discriminare in modo netto fra le componenti.

Gli oggetti della View vengono invece creati a parte come normali elementi HTML, ai quali si aggiunge un attributo `data-bind` dove specificare l'elemento del ViewModel con cui effettuare il binding e la modalità di input/output possibili (limitatamente alla possibilità di update o meno del valore, sempre che questo non sia derivato tramite risultato di una funzione).

```
// esempio di funzione ViewModel

function StudenteViewModel() {
    this.nome = ko.observable("Lorenzo");
    this.cognome = ko.observable("Cavazzi");

    this.nome_completo = ko.computed(function() {
        return this.nome() + " " + this.cognome();
    }, this);
}
// attivazione della funzione
ko.applyBindings(new AppViewModel());
```

La sintassi è in generale più semplice rispetto a Backbone e le righe di codice per la creazione del ViewModel sono sensibilmente inferiori rispetto a quelle necessarie per la definizione del Controller visto al paragrafo precedente. Tuttavia Knockout scala con difficoltà su grandi applicazioni, poiché la funzione che definisce il ViewModel, dovendo contenere tutta la logica applicativa, tende a diventare particolarmente ingarbugliata. Inoltre uno dei problemi maggiori di qui tale libreria soffre riguarda la gestione di quegli

elementi della View che devono essere creati e distrutti dinamicamente, i quali richiedono l'utilizzo di funzioni realizzate ad hoc dalla libreria per la creazione automatica di binding che mantengano la sincronia. La loro sintassi è tutt'altro che immediata, e questo è uno dei principali motivi che frena uno sviluppatore dall'adottare quest'architettura. Purtroppo il difetto appena citato è rilevabile anche per compiti ricorrenti e relativamente semplici come lo scorrimento di array di oggetti del ViewModel.

2.2.4 Limiti nello stato dell'arte

La soluzione proposta da Knockout è decisamente più veloce e immediata rispetto a quella di BackboneJS, sebbene questo determini una flessibilità minore. Tuttavia Knockout presenta una struttura piuttosto confusionaria in cui Model e ViewModel non prevedono delle strutture ben definite all'interno delle quali inserire i dati, cosa che porta la parte gestionale a una complessità elevata, rendendo la logica di business difficile da mantenere. Inoltre la View risulta piuttosto rigida, poiché i binding vengono effettuati direttamente con degli elementi HTML, i quali non possono mutare in base a condizioni di contorno come lo stato delle variabili o il contesto d'esecuzione.

Per risolvere il primo di questi problemi è nato di recente un progetto per unire Backbone e Knockout, in modo da creare una libreria che permetta di utilizzare l'approccio proposto dalla seconda con la flessibilità della prima [39]. Tuttavia non ci sono esempi di librerie che garantiscono un'elevata flessibilità della View che, nonostante la dinamicità dei contenuti che vengono sincronizzati col ViewModel, non prevede la possibilità d'intervenire sugli elementi HTML con modifiche strutturali. Questa appare come una carenza piuttosto significativa alla luce delle novità introdotte da HTML5, che dovrebbero spingere gli sviluppatori verso l'utilizzo di elementi più appropriati alle modalità d'interazione che si vuole proporre, in modo da ottenere comportamenti standard e garantire la compatibilità fra i diversi interpreti presenti nei browser e, nel prossimo futuro, direttamente nei sistemi operativi.

Infine, vista la stretta correlazione della View col ViewModel, ci si aspetta che, in molte situazioni, sia possibile derivare la prima da quest'ultimo, in modo da risparmiare la stesura di codice ripetitivo da parte dello sviluppatore. Ad esempio, se intendo esporre una data senza valore assegnato, appare logico dover inserire un elemento di input, e nel momento in cui viene assegnato un valore sarebbe opportuno trasformarlo in uno span o un altro elemento (eventualmente ancora modificabile, ma diverso da un input, che viene percepito dall'utente come qualcosa su cui bisognerebbe intervenire ancora).

CAPITOLO 3

SOLUZIONE PROPOSTA

Il lavoro svolto cerca d'inserirsi nel contesto approfondito precedentemente (cap 2), e consiste principalmente nella realizzazione di una libreria JavaScript che metta a disposizione delle API per creare applicazioni HTML5 basate sul pattern architetturale MVVM, cercando di proporre un approccio nuovo che renda la stesura di codice più immediata rispetto alle soluzioni viste nello stato dell'arte.

L'idea è quella di creare uno strumento in grado di sfruttare la stretta correlazione tra ViewModel e View, in modo da generare automaticamente quest'ultima con una serie di benefici. Primo fra tutti gli elementi della View potrebbero essere molto più flessibili rispetto a quelli definiti staticamente, così da poter essere modificati o addirittura sostituiti in base alle differenti modalità d'interazione che un'applicazione prevede a seconda del contesto d'esecuzione o dello stato delle variabili.

A fianco della libreria viene resa disponibile un'ampia documentazione, necessaria per permettere agli sviluppatori interessati di sfruttare tutte le potenzialità offerte. Adeguandoci alla necessità d'immediatezza nell'apprendimento della sintassi richiesta da una libreria emergente, si vuole fornire dei metodi semplici, e proporre una serie di tutorial interattivi così da facilitare l'apprendimento della sintassi.

3.1 Requisiti e obiettivi

Il contesto in cui ci muoviamo, come già detto più volte, è quello delle applicazioni HTML5, e in particolare lo strumento che vogliamo realizzare deve consentire di seguire un design pattern affidabile per risolvere il problema di consistenza dei dati tra interfaccia e logica applicativa. Partiamo dal presupposto che le applicazioni da sviluppare con la libreria abbiano una complessità non banale, e che quindi ci siano relazioni più o meno complesse fra i dati coinvolti nella logica applicativa; del resto, in caso contrario, i problemi di sincronismo con sarebbero gestibili senza bisogno di strumenti avanzati. Lasciamo quindi da parte le applicazioni più banali, per le esistono già alternative valide.

Per stabilire i requisiti partiamo dalle soluzioni analizzate nello stato dell'arte. Per prima cosa, tra i design pattern visti, sarebbe auspicabile affidarsi al MVVM in modo da sfruttare la vicinanza del ViewModel a quella del View, così da cercare di automatizzare la creazione di quest'ultima. Inoltre questo appare l'approccio più naturale per un ambiente di programmazione indirizzato alla gestione degli eventi come il JavaScript.

Vogliamo che la View sia un elemento dinamico, così da adattarsi ai cambiamenti a runtime e del contesto, e il miglior modo per farlo sarebbe quello di non delegare il compito della sua creazione all'utente, ma direttamente alla libreria. Un approccio di questo tipo potrebbe destare delle perplessità, poiché rischia di togliere allo sviluppatore il pieno controllo sull'output; tuttavia ci sembra l'alternativa migliore per rendere dinamica l'interfaccia utente.

Ai fini teorici la netta separazione tra Model e ViewModel è necessaria sia per definire bene i compiti delle due componenti, sia per comprendere i meccanismi che consentono la separazione tra interfaccia e logica di business. Tuttavia l'approccio proposto da Knockout, dove la suddivisione durante la stesura del codice appare più sfumata a guadagno dell'immediatezza, ci sembra particolarmente efficace. Del resto, anche se i meccanismi interni alla libreria devono rispettare le linee guida dell'architettura, questo non significa che bisogna forzare lo sviluppatore a

utilizzare una sintassi eccessivamente ingarbugliata.

Dovendo automatizzare la creazione della View è necessario che l'output appaia sempre nel modo più intuitivo possibile per ogni tipologia di elemento, andando così a coprire le esigenze di base. Bisogna comunque permettere di personalizzare velocemente il risultato che si può ottenere per rendere la libreria sufficientemente versatile, e coprire anche i bisogni più particolari dello sviluppatore.

Ovviamente gli elementi generati devono essere utilizzabili facilmente con tutti gli strumenti di manipolazione del DOM ampiamente utilizzati nella creazione di applicazioni HTML, quindi bisogna assegnare `id` e attributi di tipo `class` in modo da rendere immediati gli interventi successivi (si pensi alla manipolazione tramite le regole CSS, o alle customizzazioni che normalmente si effettuano tramite jQuery, Mootools o altri strumenti simili). Anche se diamo la possibilità allo sviluppatore di gestire questi dettagli, vogliamo che questa possibilità sia sfruttata solo per esigenze particolari, così da rendere la stesura del codice più veloce possibile.

Il concetto di binding è fondamentale nell'architettura MVVM, ma spesso è impossibile determinare automaticamente quali siano i collegamenti fra gli elementi. Questo compito viene lasciato a chi deve implementare il codice, ma solo nei casi non banali. Per questi invece cerchiamo di automatizzare anche questo processo (si pensi alla creazione dei binding per un elemento per il quale non esistono dipendenze, che può essere gestita dalla libreria).

Ricapitoliamo quanto fin qui affermato con una tabella degli obiettivi e dei requisiti:

REQUISITI	**OBIETTIVI**
Adottare l'architettura MVVM senza un approccio troppo formale	Mantenere la consistenza dei dati tra logica applicativa e interfaccia
Automatizzare la creazione degli elementi della View	Garantire flessibilità all'interfaccia in base a contesto e stato delle variabili
Automatizzare dove possibile la definizione del ViewModel	Sgravare lo sviluppatore da compiti ripetitivi e facilmente automatizzabili
Rendere la stesura del codice veloce	Rendere la libreria uno strumento realmente utile e appetibile
Adottare una sintassi semplice	

3.2 La libreria

Vediamo ora come vengono tradotti obiettivi e requisiti nell'implementazione della libreria. Per prima cosa essa si basa sulla creazione di un Model che utilizza prevalentemente oggetti istanze della classe Item. La sintassi per manipolare questi oggetti è simile a quella tipica di un linguaggio fortemente orientato agli oggetti, e si basa quindi sull'utilizzo di metodi setters e getters. All'interno degli Item si possono aggiungere attributi dei tipi standard presenti in JavaScript, con l'aggiunta di alcuni tipi utili nella gestione dei dati che tipicamente ritroviamo in un'applicazione web (ad esempio le date, oppure gli interi che in JavaScript non sono distinti in modo netto dai float).

Vengono implementate una serie di altre classi derivate dagli Item per poter gestire al meglio contenuti più avanzati come mappe, elementi multimediali, websocket. Tali oggetti verranno in seguito chiamati ItemAdvanced, mentre la loro denominazione specifica sarà data dal prefisso Item seguito da una parola che identifica il loro ambito di utilizzo, come ItemMultimedia, ItemSocket o ItemMap (par 4.1.3).

Su questi oggetti sono definiti una serie di metodi per personalizzare il comportamento dei corrispondenti elementi del ViewModel e della View. I primi vengono creati quando il documento è completamente interpretato, in modo da avere già in memoria tutti i parametri definiti negli Item; i secondi vengono generati appena dopo questi, poiché derivano direttamente da essi.

I metodi e gli attributi che influiscono sui comportamenti dei 3 diversi componenti sono separati a livello logico, ma questa divisione viene resa trasparente allo sviluppatore. Si ritiene che essa potrebbe solo rappresentare un'inutile complicazione, mentre si preferisce evidenziare la differenza tra metodi utilizzabili nella fase di runtime rispetto a quelli per la fase d'interpretazione del codice. I motivi di questa scelta riguardano principalmente la possibilità di separare gli elementi statici da quelli generati a realtime, in modo da favorire l'eventuale salvataggio degli aggiornamenti in remoto, e anche il mantenimento delle relazioni all'interno del ViewModel, di cui parliamo più nel dettaglio nel prossimo capito.

3.2.1 Meccanismi di base e approccio MVVM

Partiamo da una descrizione generale dove evidenziamo come viene implementato l'approccio MVVM e come viene gestita la separazione dei 3 componenti dalla libreria. Gli oggetti che lo sviluppatore crea tramite essa sono istanze di Item (o ItemAdvanced) dei quali è possibile definire una serie di attributi. Alcuni di essi sono fondamentali, e riguardano il tipo di dato trattato, il valore contenuto e altri dettagli che costituiscono il Model, mentre altri sono opzionali e definiscono meglio il comportamento degli elementi a runtime.

Ogni sotto-oggetto degli Item e degli ItemAdvanced, una volta terminata l'interpretazione del documento, va a generare in automatico il suo corrispondente nella ViewModel e successivamente nella View. La tipologia di elemento viene scelta in base al contenuto del ViewModel e ad eventuali variabili di contorno determinabili coi metodi da utilizzare sugli Item e gli ItemAdvanced.

```
// Esempio codice JavaScript per creare e personalizzare un Item

var studente = new Item();
studente.addString("nome", "cognome");
studente.addDate("nascita").addNumber("eta");
studente.setLocalStorage(true);
studente.setValue({nome: "Lorenzo", cognome: "Cavazzi"});

studente.nome.setModifiable(false);
studente.eta.setValue(24).setIsInteger(true).setMin(0);

studente.addString("fullName");
studente.fullName.setBinding(function(){
    return studente.nome.get()+" "+studente.cognome.get();
});

studente.fullName.setLabel("nome completo");
```

Nell'esempio di codice proposto vediamo come nelle prime 5 righe viene definito il Model. L'Item viene arricchito di attributi personalizzabili (da ora in avanti li chiamiamo sObject), poi vengono aggiunti dettagli sullo stato iniziale delle variabili e sul metodo di memorizzazione. Dopo questa fase troviamo 2 righe che definiscono il comportamento degli sObject. In questo caso stiamo

intervenendo sugli oggetti del ViewModel (ad esempio `studente.nome` viene reso non modificabile, ma questo riguarda l'interazione dell'interfaccia utente e non un blocco totale di assegnamento via codice che non sarebbe possibile). Anche la funzione `setBinding()` determina comportamenti nel ViewModel, in particolare lega il valore di uno sObject a quello di altri, garantendo la consistenza ogni volta che ci sono aggiornamenti da parte delle fonti. Con questi strumenti è possibile creare dei binding senza utilizzare una sintassi particolare come avviene nel caso di Knockout. L'ultima riga di codice infine personalizza un aspetto della View, in particolare l'etichetta che viene assegnata a un elemento che contiene tipi di dato semplici (stringhe, numeri, date o boolean).

Un osservatore attento avrà notato che nella prima parte i metodi vengono invocati direttamente sull'Item, mentre nel secondo caso si interviene sui suoi sotto-oggetti. Questo segna in qualche modo la differenza fra intervento sul Model e sul ViewModel, ma non è enfatizzata una separazione in modo da rendere il programmatore ignaro della profonda differenza nella modalità di gestione delle due da parte della libreria (de resto in altri casi tale separazione è meno percepibile, come vedremo per gli ItemAdvanced nel paragrafo 4.1.3).

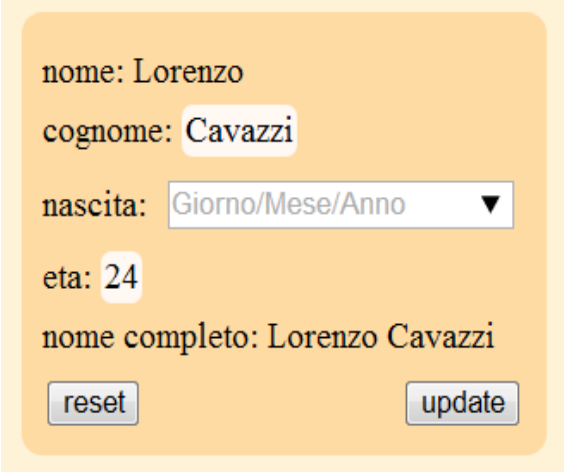

Figura 3.1: output del codice presentato sopra

Infine in figure 3.1 vediamo l'output nel browser del codice d'esempio. Si notino le differenze tra gli elementi con valori non assegnati (`input`), quelli assegnati su cui è possibile intervenire (`span` editabili) e sugli altri non modificabili (`span` semplici).

3.2.2 Scelte implementative

Quando parliamo d'interfaccia intendiamo quella parte attraverso il quale l'utente può interagire con l'applicazione, sia visualizzando l'output che tramite input. Nel caso dell'architettura MVVM essa coincide quasi perfettamente con la View. Pur avendo già descritto accuratamente questo componente, potrebbe esserci ancora un po' di confusione a riguardo per chi non conosce bene la programmazione HTML5. Nello specifico per View intendiamo il contenuto del documento HTML, quindi gli elementi, ma non il loro stile di visualizzazione. Ogni tipo di elemento ha delle peculiarità che lo rendono diverso dagli altri, ma la visualizzazione vera e propria passa attraverso le regole CSS (par 2.1.3) che di fatto determinano l'output finale. Il semplice cambiamento delle regole attuate porta solitamente a un impatto visivo completamente diverso.

La libreria, in accordo con le regole basilari di buona programmazione HTML [40], non si occupa di gestire l'impaginazione dei contenuti che genera. Del resto della grafica dell'interfaccia solitamente non si occupa il programmatore della logica dell'applicazione, che è il target della libreria, ma piuttosto un designer o un grafico. Nella creazione di tali contenuti bisogna comunque tener conto di come poter agevolare questo successivo lavoro di resa grafica.

I documenti CSS si basano sulla selezione di elementi del DOM per poterne manipolare l'aspetto [41], e tipicamente si utilizzano i nomi dei tag, gli id e le classi. L'utilizzo di id e classi è il metodo preferenziale per selezionare elementi specifici o tipologie di elementi che svolgono una funzione comune. La libreria assegna automaticamente un id a ogni elemento basandosi sul nome della variabile che gli corrisponde nel Model in modo che, ad esempio, l'elemento HTML corrispondente all'attributo `dataNascita` dell'Item `persona` abbia assegnato l'id `persona_dataNascita`. Allo stesso modo vengono assegnate alcune classi standard a elementi con una funzione ben precisa. Per esempio i pulsanti di update sono della classe `update_button`, mentre le sezioni coi contenuti degli Item saranno della classe `easyHTML_item`.

Un approccio del genere semplifica anche eventuali modifiche effettuate con jQuery o altre librerie che si basano sulla selezione degli elementi tramite DOM.

Abbiamo deciso di disaccoppiare il più possibile la parte di costruzione del Model e del ViewModel dal posizionamento dei contenuti all'interno del documento, ovvero dell'inserimento del corrispettivo elemento nella View. Per determinare la posizione degli elementi si utilizzano delle funzioni JavaScript da posizionare inline nel documento HTML. Come parametro viene passato l'id dell'elemento generato che corrisponde al nome della variabile utilizzata nel Model.

```
// Posizionamento di un elemento nel documento

</script>
    H5.here("studente", "universita");
    H5.inside("mappaUni", "universita");
</script>
```

Questa scelta è compatibile con quella precedentemente descritta per agevolare il lavoro dei grafici, poiché il posizionamento degli elementi all'interno della pagina è solitamente basato sul flusso naturale dei dati per comporre l'interfaccia (sebbene dall'introduzione di CSS3 questa parte dovrebbe essere effettuata con regole di stilizzazione, è difficile che quest'abitudine possa cambiare a breve). Inoltre ciò consente di disaccoppiare la View dagli elementi di Model e ViewModel, cosa fondamentale considerando che la prima viene generata solo una volta interpretato tutto il documento e quindi a runtime.

Un'altra scelta importante riguarda la decisione di utilizzare gli elementi HTML5 che sono arrivati almeno allo stato di Last Call. Sebbene questo crei dei problemi di compatibilità (par 2.1.5), riteniamo che fornire supporto ai browser obsoleti (sebbene rappresentino ancor oggi una percentuale rilevante [42]), rappresenti un limite troppo forte; la carenza di strumenti dettata da una scelta del genere finirebbe col determinare la creazione di una libreria incompleta e non in grado di soddisfare le esigenze di chi sviluppa applicazioni di una certa complessità. Del resto riteniamo che senza le nuove funzionalità sia difficile creare applicazioni moderne.

CAPITOLO 4

IMPLEMENTAZIONE

Passiamo ora a una descrizione più dettagliata della libreria (che da qui in avanti chiameremo col nome di easyHTML) e in particolare del meccanismo con il quale viene implementato il pattern MVVM. Vengono approfondite le funzioni più significative messe a disposizione degli sviluppatori, e sono fornite argomentazioni sulle scelte effettuate.

Presentiamo prima il Model poiché definisce le risorse a disposizione del programmatore sulle quale è possibile lavorare, per poi passare al ViewModel e alla View, che vengono in parte generate automaticamente, ma sulle quali si può intervenire in diversi modi per personalizzare il risultato finale. Si tenga presente che la separazione netta tra Model, ViewModel e View, pur essendo molto utile nella descrizione teorica, a livello pratico si è rivelata difficile da attuare in modo rigoroso. Sebbene ciò fosse possibile, l'utilizzo della libreria sarebbe diventato (in alcuni casi) poco intuitivo, per cui abbiamo scelto di essere meno rigidi nella separazione tra i componenti, così da privilegiare l'immediatezza nella stesura del codice da parte dello sviluppatore.

4.1 Model e manipolazione dei dati

Nell'approccio MVVM il Model fornisce un metodo per accedere ai dati utili all'applicazione [43]. Questi, nel caso di un'applicazione HTML5, possono essere intesi come la struttura dei dati e il loro valore di partenza. Oltre a ciò, nel Model rientrano anche una serie di dati accessori che il ViewModel può utilizzare per esporli nella View o per determinare alcune peculiarità della stessa [44].

EasyHTML rende disponibili delle classi JavaScript da utilizzare per istanziare nuovi oggetti. In realtà JavaScript non utilizza delle classi vere e proprie, ma consente di creare delle funzioni all'interno delle quali si possa utilizzare la parola chiave `this` per differenziare il contenuto di eventuali nuove istanze, che di fatto sono oggetti clone creati attraverso la parola chiave `new` [45]. Questi oggetti, a seconda della tipologia, forniscono dei metodi per aggiungere contenuti (una stringa di testo, un numero, delle coordinate geografiche...) e degli attributi che determinano alcuni dettagli sulla tipologia di dato da gestire o eventuali informazioni aggiuntive che, a seconda del contesto, possono essere visualizzate nella View o utilizzate dal ViewModel per la validazione dei dati e la creazione di binding (par 4.2.1).

La logica dell'applicazione è creata in gran parte assieme al Model, ma allo stesso tempo vengono create una serie di funzioni che vanno a definire il ViewModel, in particolare per quanto riguarda la gestione di binding diversi da quelli standard. A differenza dell'approccio utilizzato nelle librerie viste in precedenza (par 2.2), la validazione dei dati viene quasi completamente automatizzata, per cui lo sviluppatore può limitarsi a definire alcuni attributi (eccetto il caso in cui desideri accettare solo valori particolari). Oltre alle tipologie tipiche, come boolean, number e string, ne vengono introdotte di nuove, ad esempio le date e le coordinate geografiche, in modo da semplificare la gestione di elementi che, in ambito web, vengono utilizzati spesso.

4.1.1 Item

La classe di base introdotta da easyHTML è Item. Questa può essere vista come un Object particolare per il quale sono definiti una serie di attributi che vengono utilizzati per la creazione del ViewModel e della View, e all'interno di esso si possono aggiungere diversi contenuti.

Tutti i dati da gestire nel Model devono essere contenuti all'interno di oggetti istanze di Item (da qui in avanti si indicheranno semplicemente come oggetti Item). A un Item non deve per forza corrispondere un elemento nella View, ma per garantire la consistenza dei dati tutto ciò che è visualizzato nella View deve essere memorizzato all'interno di un Item. Gli attributi gli vengono aggiunti utilizzando i metodi `addString()`, `addNumber()`, `addDate()` o `addBoolean()`, che inseriscono degli oggetti creati per contenere un eventuale valore dello stato della variabile o un riferimento a una variabile esterna, insieme a una serie di altri metadati (par 4.1.2).

Per visualizzare un Item è necessario inserire il codice `here("ItemName")` all'interno del documento HTML nella posizione desiderata. `ItemName` è il nome dell'oggetto Item da inserire. L'utilizzo del nome della variabile al posto della variabile stessa come argomento della funzione `here()` potrebbe destare qualche perplessità. Effettivamente a un oggetto può corrispondere più di un nome, ma con questo approccio abbiamo alcuni benefici nell'implementazione del design pattern. Per prima cosa avviciniamo il Model alla View, poiché all'oggetto JavaScript `ItemName` la libreria fa corrispondere l'id dell'elemento HTML, rendendo più facile sia la generazione automatica degli elementi, sia il compito di chi deve rifinire l'interfaccia utente. Inoltre disaccoppiamo il posizionamento degli oggetti dalla loro creazione, evitando errori dovuti all'utilizzo di risorse non ancora disponibili. Del resto la funzione `here()` non posiziona elementi del Model, ma elementi della View (anche se in realtà tra i due c'è stretta correlazione), quindi risulta più sensato passare l'id dell'elemento HTML (par 4.2.3). Nel caso in cui lo stesso Item fosse associato a 2 variabili differenti, nel ViewModel verrebbero generati 2 elementi differenti, ma solo quello posizionato con la funzione `here()` nel documento avrebbe un corrispondente nella View.

Per clonare un Item viene reso disponibile il metodo `clone()`, che riceve in input un altro Item (o il suo nome) e ne copia tutti gli attributi. Lo stesso effetto si ottiene passando il parametro al costruttore nel momento in cui un nuovo oggetto viene istanziato.

Per rendere permanenti le modifiche effettuate a runtime si possono utilizzare diversi metodi. Il più semplice sfrutta il Local Storage introdotto da HTML5. Esso è un oggetto JavaScript che può contenere delle variabili con associati valori di tipo stringa [46]. Per aggirare questa limitazione si può serializzare un oggetto in una stringa JSON (par 4.3.2), così da salvare anche contenuti più elaborati (si tenga però presente che ogni dominio ha uno spazio limitato per archiviare dati, per cui non è possibile salvare immagini, video o altri contenuti multimediali). Ponendo l'attributo `localStorage` a `true` con il metodo `setLocalStorage(true)`, la libreria si occuperà di salvare automaticamente in locale i dati, e ogni volta che la pagina verrà carica, sarà ripristinato l'ultimo stato memorizzato. Vengono salvati solo i contenuti delle variabili, non eventuali modifiche strutturali effettuate a runtime (par 4.3.2).

Spesso nella realizzazione delle applicazioni è necessario salvare parte dei dati su un server remoto. EasyHTML mette a disposizione il metodo `onUpdate()` per definire funzioni che gestiscono l'invio con un metodo qualunque. Se ci si affida invece ai Socket, viene messo a disposizione l'oggetto ItemSocket che automatizza ricezione e invio dei dati (par 4.2.2).

Ci sono poi una serie di altri metodi che servono per aggiungere metadati che easyHTML utilizza per generare il ViewModel o per intervenire su alcuni elementi della View. La maggior parte di questi attributi sono vuoti di default (non determinano quindi alcun effetto se non assegnati), e le modalità per modificarli con i relativi effetti prodotti vengono descritti nel dettaglio nel paragrafo 4.2.3. Per ulteriori approfondimenti sui metodi da invocare sugli Item si rimanda all'Appendice A, dove viene riportata una reference più dettagliata.

```
// Esempio codice JavaScript per creare un Item

var studente = new Item();
studente.addString("nome", "cognome");
studente.addDate("nascita").addNumber("eta");
studente.setLocalStorage(true);
studente2 = new Item(studente);
```

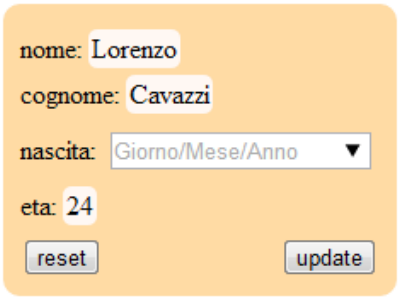

Figura 4.1: output del codice d'esempio di creazione di un Item

4.1.2 sObject

Con il nome di sObject indichiamo una classe astratta da cui ereditano delle sottoclassi specializzate nel contenere determinate tipologie di dato. Queste sottoclassi sono `sNumber`, `sString`, `sDate` e `sBoolean` che contengono rispettivamente dati di tipo number, string, date e boolean. In sostanza queste sono le tipologie di dato gestite dagli Item, a cui vanno aggiunte quelle gestite dagli ItemAdvanced (par 4.1.3).

Gli oggetti di tipo sObject sono istanziati come attributi degli Item, e la sintassi utilizzata è quella vista nel paragrafo precedente, ovvero `myItem.ddNumber()`, `myItem.addString()`, e così via. Possiamo considerare questi oggetti come i tipi di base gestiti dal Model. I valori contenuti negli sObject possono essere assegnati direttamente con il metodo `setValue()` o definiti in fase di creazione dell'oggetto con la sintassi abbreviata `addNumber("name:value")`. In alternativa il loro contenuto può essere collegato tramite binding nel ViewModel a un altro sObject, o a una funzione dove si possono utilizzare valori di uno o più sObject (par 4.2.1), o ancora valori provenienti da fonti remote e gestite dagli ItemSocket (par 4.2.2). In tutti questi casi viene garantita la consistenza dei dati per ogni eventuale cambiamento da parte di una o più fonti.

I binding possono essere effettuati anche verso normali variabili JavaScript, ma in questo caso non si garantisce la consistenza in ogni

situazione. Per aggirare questo ostacolo è sufficiente pre-caricare i dati collegandoli a un Item, oppure assicurarsi che un evento di update venga lanciato sull'Item ogni volta che la variabile sorgente viene modificata (par 4.3.4).

Il valore memorizzato negli sObject viene salvato nel Model ogni volta che il ViewModel propaga una modifica (se è prevista la permanenza delle modifiche tramite View), e viceversa viene propagato al ViewModel ogni volta che s'interviene sul Model. Tale comportamento è valido a runtime, mentre nelle fasi precedenti il ViewModel non è ancora stato generato, per cui eventuali variazioni non hanno ancora bisogno di meccanismi di sincronizzazione. Per estrarre il valore corrente dal Model (o quello del ViewModel una volta a runtime) si utilizza il metodo `get()`, come da prassi nei linguaggi di programmazione a oggetti. Il dato viene recuperato direttamente dalla struttura locale, oppure dal Local Storage se la variabile di controllo dell'Item è impostata a *true*, o ancora dalla fonte esterna (eventualmente remota) in caso di binding.

Abbiamo già visto che il Model, nel pattern MVVM, può ospitare eventuali attributi di contorno che hanno influenza sulla View, pur non modificando l'effettivo valore delle variabili [44]. Questi attributi possono essere interpretati come metadati che definiscono meglio la tipologia del contenuto. Ad esempio un sObject può essere mutabile o immutabile, e per memorizzare tale informazione è disponibile il metodo `setModifiable(true/false)`. Un sObject impostato come non modificabile non è bloccato alle modifiche via codice (del resto avrebbe poco senso, poiché in qualche modo dovrò assegnare qualche significativo), ma viene interpretato in modo differente dal ViewModel, impedendo di fatto la possibilità di effettuare modifiche nell'interfaccia da parte dell'utente, o comunque che esse vengano propagate fino a cambiare lo stato effettivo della variabile. In pratica, una volta caricato il valore dal Model, questo risulta bloccato.

Altri attributi influiscono in modo diretto già sul Model. Per esempio un sNumber può essere di tipo intero o non intero, che corrisponde ai tipi `Integer` e `Float` comunemente usati nei linguaggi di programmazione, ma non gestiti in modo nativo da JavaScript. Posso stabilire anche dei valori minimi e massimi all'interno del quale deve trovarsi il valore per essere

significativo (si pensi a un numero che memorizza l'età, o un parametro di scelta fra alcuni elementi disponibili). In questi casi i metadati influenzano anche il comportamento del Model, poiché non è mai possibile assegnare dei valori fuori range senza generare un'eccezione. Altri metodi che influenzano il ViewModel e la View verranno descritti più avanti (par 4.2).

```
// Esempio codice JavaScript per creare e personalizzare gli sObject

var studente = new Item();
studente.addString("nome", "cognome").addNumber("eta");

studente.nome.setValue("Lorenzo").setModifiable(false);
studente.eta.setValue(24).setIsInteger(true).setMin(0);
studente.setValue({cognome: "Cavazzi"});

console.log("studente: ", studente.nome.get() + " " +
studente.cognome.get());
```

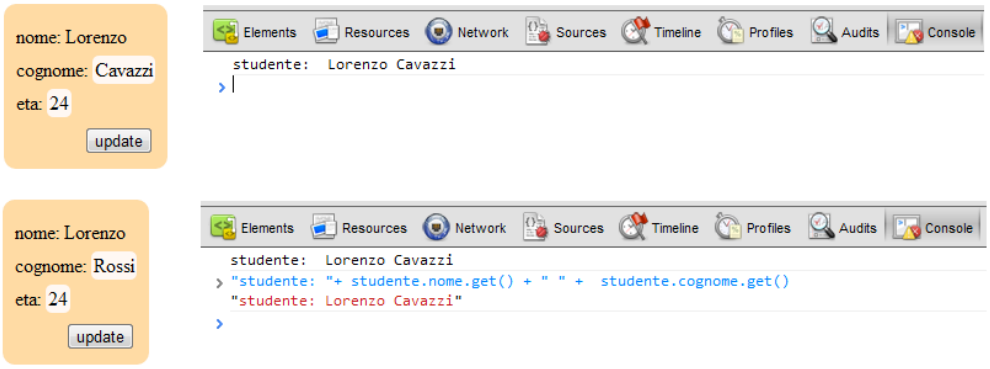

Figura 4.2: cambiamento dello stato del ViewModel a seguito di modifiche alla View

4.1.3 ItemAdvanced

Gli ItemAdvanced contengono dati simili a quelli degli Item, ma sono specializzati nella gestione di oggetti particolari. La classe ItemAdvanced è astratta, e contiene una serie di sottoclassi: `ItemMap`, `ItemImage` e `ItemMultimedia`. Ognuno di essi, come lascia intuire il nome, gestisce una tipologia specifica di elementi. In questo senso troviamo delle analogie anche

con gli sObject, ma abbiamo deciso di tenerli separati da questi ultimi sia per la quantità d'informazione che gestiscono, sia per dare la possibilità di posizionarli in modo indipendente dagli Item. Inoltre, considerando come vengono gestiti nella View gli eventi scatenati dall'interazione dell'utente (par 4.2.3), trattare questi oggetti in modo analogo agli sObject avrebbe determinato un comportamento anomalo (ad esempio video che ripartono dall'inizio o reset delle mappe interattive).

Ogni ItemAdvanced può avere dei sottotipi; ad esempio gli ItemMap si occupano genericamente di gestire le coordinate, ma possono avere comportamenti che si differenziano molto tra di loro. Si possono indicare singole posizioni geografiche, e in questo verrà memorizzato il nome della località (o le relative coordinate), oppure si possono gestire percorsi o posizioni multiple, e in questo caso verrà salvata una lista di elementi.

La visualizzazione può essere più o meno semplificata, passando da indicazioni testuali per arrivare a mappe interattive. Allo stesso modo un ItemMultimedia può gestire flussi audio o video, e un ItemImage può gestire singole immagini o slider e caroselli. Trattare tutti i dettagli degli ItemAdvanced richiederebbe molto tempo, quindi ci concentreremo solo su alcuni meccanismi generali. Per i lettori più interessati si rimanda all'appendice A per una descrizione più dettagliata di oggetti e metodi disponibili.

Un ItemAdvanced ha delle fonti associate, che possono essere l'URL delle immagini o dell'elemento multimediale, oppure le coordinate geografiche o un nome che identifica la posizione di una località. Inoltre è necessario specificare una tipologia per ogni ItemAdvanced. Per le mappe possiamo scegliere tra coordinate, single, multi e path che gestiscono rispettivamente zone identificate da coordinate, dal nome della posizione, da una lista di nome nomi o da una serie di coordinate. Per gli ItemMultimedia si può scegliere tra audio o video, ed è necessario impostare l'URL di riferimento della (o delle) sorgente dati. E' possibile specificare meglio il comportamento con i metodi `setCommand()` e `setDisplay()` che determinano le possibilità di controllo associate all'elemento multimediale; ad esempio si può permettere o negare di spostarsi più avanti nel flusso video, o decidere se visualizzare o meno i comandi per regolare il volume e altri parametri per la riproduzione. Gli ItemImage invece gestiscono immagini. Nel caso di immagine singola si

imposta solo l'URL, mentre selezionando le tipologie carousel o multi è possibile creare slider. In quest'ultimo caso si può intervenire sul delay.

```
// Esempio codice JavaScript per creare diverse tipologie di
// ItemAdvanced

var ItemMap1 = new ItemMap();
ItemMap1.setCommand(true).setZoom(15);
var ItemMap2 = new ItemMap();
ItemMap2.setType("multi").setZoom(6);
ItemMap2.addPosition("Milano", "Bologna", "Venezia", "Trieste");

var ItemImage1 = new ItemImage();
ItemImage1.setURL("other/logo.png");
var ItemImage2 = new ItemImage();
ItemImage2.setType("carousel");
ItemImage2.addURL("other/logo.png", "other/polimi.jpg");

var ItemMultimedia1 = new ItemMultimedia();
ItemMultimedia1.setLabel("video").setType("video");
ItemMultimedia1.setURL("other/movie.mp4");
var ItemMultimedia2 = new ItemMultimedia();
ItemMultimedia2.setLabel("audio").setType("audio");
ItemMultimedia2.setURL("other/song.mp3");
```

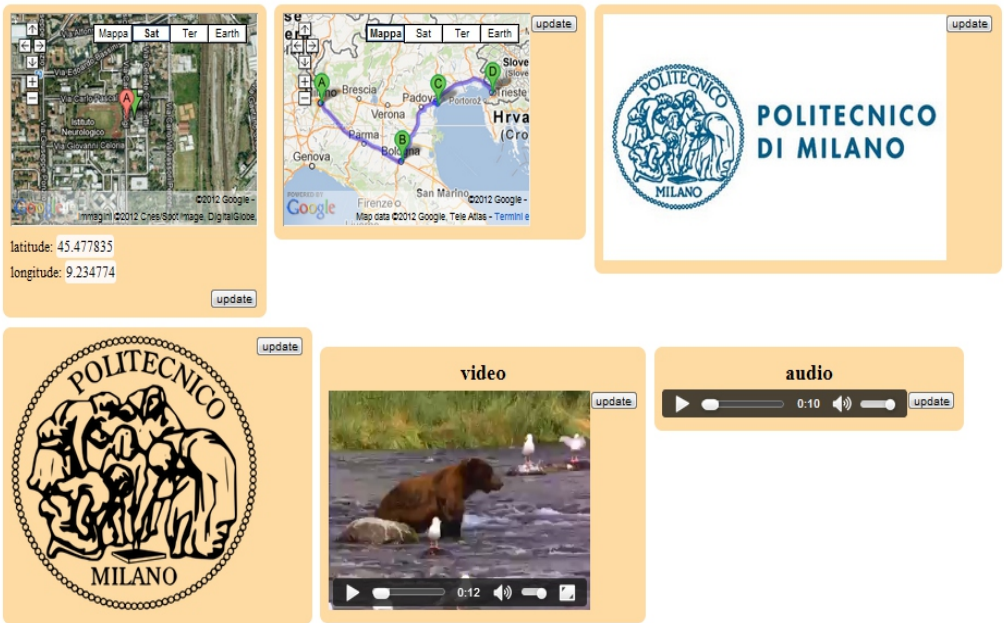

Figura 4.3: output nella View di tipologie diverse di ItemAdvanced

4.2 ViewModel e View

Il ViewModel e la View sono generati, almeno in parte, automaticamente da easyHTML. Per fare questo la libreria parte dal Model e, una volta che il documento è stato completamente interpretato, viene scatenato un evento per la generazione delle 2 componenti. Per ogni Item, ItemAdvanced e sObject verrà creato un oggetto corrispondente nel ViewModel. Grazie alla struttura di Item e ItemAdvanced, a ogni modifica effettuata direttamente sui dati del Model, utilizzando i metodi setters a disposizione, viene propagata un notifica nel caso in cui ci si trovi a runtime (ovvero se per l'utente è già disponibile l'output e le interazioni). Il ViewModel si occupa di recuperare il nuovo valore e, se differente da quello attualmente memorizzato, provvederà all'update.

Una volta costruito che questo viene costruito e gli sono associati gli eventi del Model, è il momento di generare automaticamente la View. Non tutti gli oggetti del Model devono necessariamente avere un corrispondente elemento HTML, magari perchè non si vuole permettere alcuna interazione con l'utente o perchè non sarebbe significativo esporre i dati che contiene. Per questo motivo vengono creati elementi solo per quegli Item e ItemAdvanced per cui è stata definita una posizione nel documento con le funzioni `here()` o `inside()`. Gli elementi così creati vengono posizionati e, ad ognuno di essi, viene associato un evento di update che indica un qualche cambiamento effettuato dall'utente tramite interfaccia, e che dovrà essere gestito dal ViewModel in modo simile a quanto descritto sopra per il sincronismo con il Model. Questa volta però il cambiamento di stato dovrà essere propagato dalla View fin verso il Model.

Mantenere la consistenza per un elemento della View che corrisponde direttamente a un elemento del Model è semplice, ma i binding tra sObject, quelli verso oggetti in remoto e quelli verso funzioni che coinvolgo più sObject comportano un maggior numero di problemi. Li approfondiamo più avanti (par 4.3) dopo aver visto le diverse tipologie di binding.

Per il momento si noti come uno dei principi fondamentali del MVVM [44] sia rispettato: Il Model, pur contenendo dei metadati aggiuntivi, è inconsapevole dell'esistenza degli altri 2 componenti, e la stessa cosa si può

dire della View. Al contrario il ViewModel comunica sia con il Model che con la View, andando a modificare entrambi e assumendo quel ruolo di controllore che si occupa di mantiene il sincronismo tra le parti e di tradurre e validare i dati ogni volta che sia necessario (fig 4.4).

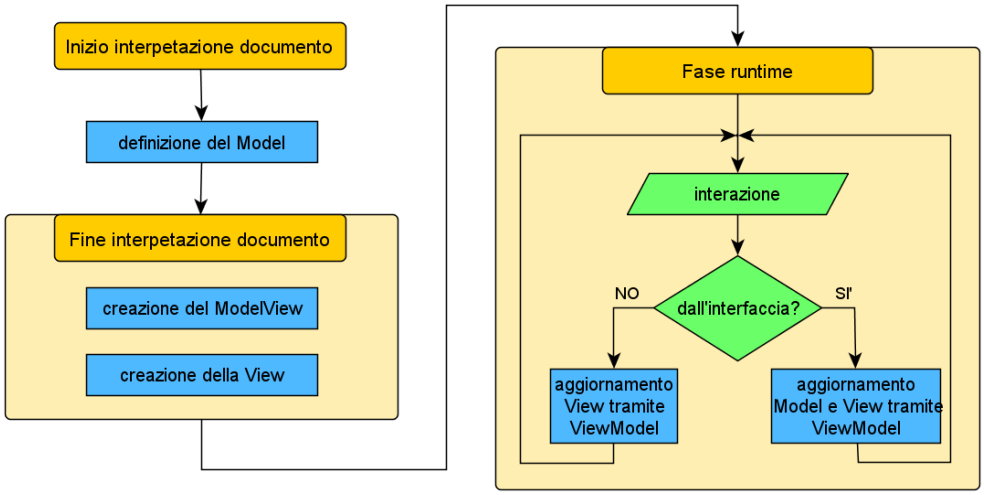

Figura 4.4: diagramma semplificato delle fasi di creazione e sincronizzazione dei componenti MVVM in easyHTML

4.2.1 Binding

Per binding si intende un collegamento fra dati che crea una dipendenza, vincolando i cambiamenti da parte di una delle due fonti a quella dell'altra. Nel design pattern MVVM il termine binding indica un legame tra risorse del Model e ViewModel, o tra quest'ultimo e la View [47]. Questi legami vengono creati, in buon parte, in modo automatico dalla libreria. In generale possiamo parlare anche di binding tra risorse del Model e della View, tenendo però presente che quest'ultimo non è diretto ma richiede un doppio passaggio attraverso il controllore.

Il caso più semplice di binding è quello 1 a 1 fra un elemento del Model e una sua rappresentazione nella View. Si considera come singolo elemento del Model un sObject o un ItemAdvanced, poiché ognuno di essi gestisce un dato atomico, mentre un Item è un elemento multiplo, e ogni suo sottoelemento

può avere binding customizzati. Il ViewModel, in questo caso, utilizza direttamente il Model per memorizzare il dato più aggiornato, servendosi quindi il valore della variabile di riferimento e intervenendo direttamente su quella in caso di aggiornamenti. Da un punto di vista logico lo stato della variabile è presente sia all'interno del ViewModel che del Model, ma la sua sincronizzazione non è un problema poiché, a livello implementativo, l'oggetto che contiene i valori è lo stesso. Viene quindi creato un elemento HTML (o più d'uno in alcuni casi) e posizionato nella View (par 4.2.3), e un evento responsabile sia della sincronizzazione degli update provenienti sia da parte della View che dal Model. Sfruttando i metadati salvati in quest'ultimo viene creata la funzione di conversione e (ove necessario) di validazione dei dati. In fig 4.5 schematizziamo i meccanismi fin qui descritti, evidenziando l'effettiva residenza dei dati con colorazioni differenti per le tre componenti del pattern MVVM.

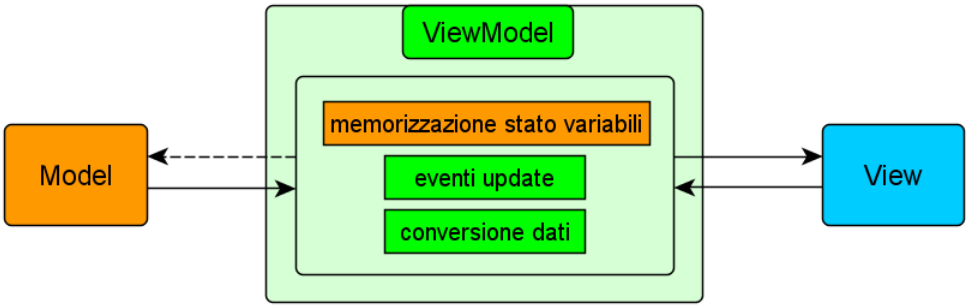

Figura 4.5: gestione nel ViewModel dei binding diretti generati in automatico

Capita spesso di voler visualizzare nella View degli elementi del Model in più posizioni differenti senza replicare la fonte. Ad esempio potrei avere un Item `persona` e un Item `esame_di_guida` che visualizza nel campo `candidato` l'attributo `codice_fiscale` di `persona`. In questo caso voglio poter creare un binding tra `esame_di_guida.candidato` e `persona.codice_fiscale`. Per fare questo viene messo a disposizione il metodo `setBinding()`, che può essere invocato su un sObject, passando come argomento un altro sObject valido. Il lettore più attento si starà chiedendo perchè invocare tale metodo su un sObject se vogliamo creare un binding tra un elemento del Model e uno della View, non tra 2 elementi del

Model. In realtà, per generare automaticamente elementi nella View, è necessario che questi esistano anche nel Model, quindi dobbiamo prima creare un sObject (al quale non verrà mai assegnato un valore esplicitamente) e poi collegarlo a un altro sObject di riferimento. Sebbene a livello teorico questa sembri una forzatura, da un punto di vista pratico tale approccio risulta abbastanza naturale per un programmatore che utilizza easyHTML, poiché si troverà così a generare tutti gli elementi della View tramite JavaScript, senza dover scrivere righe di codice HTML. Possiamo affermare che il Model risulta completo rispetto alla View. Ciò non significa che, viceversa, tutti gli elementi di un Item che hanno una posizione nel documento debbano per forza apparire nella View. E' sufficiente settare il parametro `invisible` con il metodo `setInvisible(true)` su qualunque oggetto del Model per escluderlo in ogni caso dall'interfaccia utente.

```
// Esempio creazione di binding
var persona = new Item();
var esame_di_guida = new Item();
persona.addString("codice_fiscale:CVZLNZ87C15I829S");
// creo il binding
esame_di_guida.addString("candidato");
esame_di_guida.candidato.setBinding(persona.codice_fiscale);
// escludo codice_fiscale dalla visualizzazione diretta
persona.codice_fiscale.setInvisible(true);
```

Analizziamo il ViewModel in questo esempio. L'elemento del Model che è legato a un altro tramite binding, non avrà nel ViewModel un riferimento allo stato della variabile originale, poiché questa sarà vuoto, visto che risiede già in un altro oggetto. Ci sarà invece un riferimento allo stato della variabile dell'elemento corrispondente (fig 4.6), mentre verranno creati gli eventi che gestiscono l'update (ma solo da View a ViewModel) e quelli per la sincronia locale. Da un punto di vista teorico l'elemento del Model serve unicamente alla generazione dell'elemento nella View e a fornire un riferimento per gestire correttamente quest'ultimo nel ViewModel. Lo stato della variabile e i metadati per la corretta gestione della consistenza saranno presi tutti dallo sObject referenziato. Lo schema potrebbe essere semplificato come quello in figura 4.7, rendendo le parti tratteggiate utili a costruire correttamente ViewModel e View, ma del tutto superflue una volta a runtime.

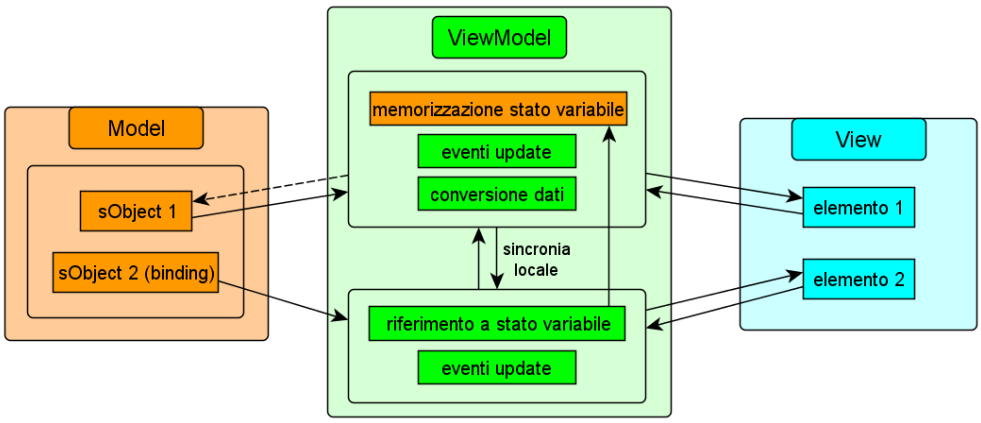

Figura 4.6: gestione nel ViewModel dei binding tra elementi

La possibilità di definire un binding di questo tipo arricchisce notevolmente la flessibilità della View, ma creare una corrispondenza esatta è spesso troppo semplice per le finalità dell'applicazione. Per questo l'argomento della funzione `setBinding()` può essere anche una funzione che ritorni un risultato coerente con la tipologia di contenuto della variabile.

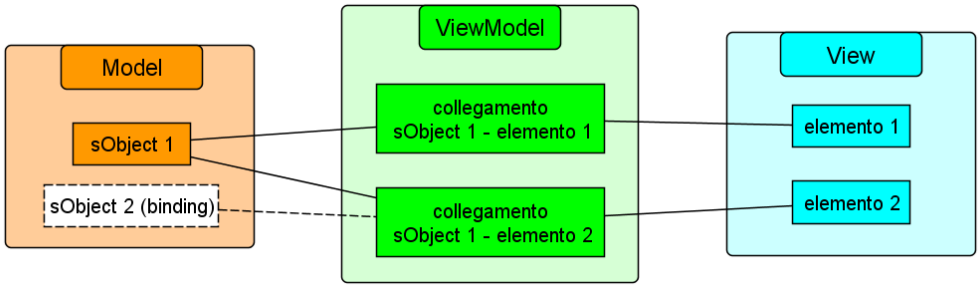

Figura 4.7: schema semplificato di binding tra elementi

L'eventuale uso di valori presi da altri sObject o ItemAdvanced all'interno di questa funzione viene rilevato dalla libreria e, nella definizione del ViewModel, vengono create delle associazioni con i corrispondenti handler, in modo da garantire la consistenza dei dati nel caso di modifiche a una fonte di dati qualsiasi.

```
// Esempio di binding tramite funzione
var persona = new Item();
var esame_di_guida = new Item();
persona.addString("nome:Lorenzo", "cognome:Cavazzi");
// creo il binding
```

```
esame_di_guida.addString("candidato");
esame_di_guida.candidato.setBinding(function(){
    return persona.nome.get()+" "+persona.cognome.get();
});
```

Prendendo spunto dal codice d'esempio riportato sopra, l'handler dello sObject `candidato` che verrà creato nel ViewModel sarà collegato a entrambe gli oggetti utilizzati nella funzione, e il cambiamento dello stato delle variabili di riferimento determinerà l'aggiornamento anche dello stato di `candidato`, per il quale verrà calcolato il valore aggiornato eseguendo nuovamente la funzione. In figura 4.5 vediamo schematizzata la struttura creata nel caso dell'esempio di codice visto sopra.

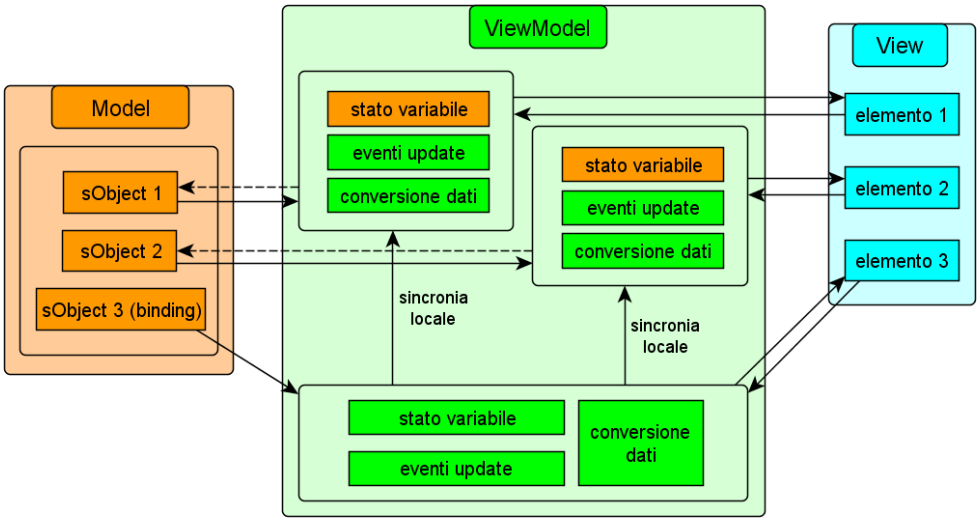

Figura 4.8: gestione nel ViewModel dei binding tramite funzione

In questo caso l'elemento del Model non è un token come per i binding fra sObject, ma partecipa in modo più importante alla creazione dei corrispondenti elementi nel ViewModel e nella View. Per prima cosa i metadati vengono utilizzati direttamente, e non si fa riferimento a quelli degli altri oggetti coinvolti. Del resto non avrebbe senso ereditare attributi come `isInteger` e `Min` definiti in un contesto ben preciso nel caso si voglia calcolare una funzione che può avere dominio del tutto differente, e ancora meno se si utilizzano dati di tipi differenti (ad esempio se si visualizza un numero che viene calcolato sfruttando il valore di verità di un sBoolean). Non essendoci una corrispondenza diretta è necessario memorizzare in locale lo stato della

variabile da visualizzare, e la sincronia locale è monodirezionale (gli oggetti target dei binding generano cambiamenti ma non li subiscono). Alcuni comportamenti a runtime dell'elemento della View sono bloccati: ad esempio non è possibile modificare direttamente il valore, poiché una funzione definita senza porre limitazioni non è di solito banalmente invertibile. Si noti anche come cambia di conseguenza lo schema semplificato in fig 4.9.

EasyHTML, per determinare gli oggetti utilizzati nella funzione, scansiona il codice verificando se vengono trovate coincidenze con un'espressione regolare del tipo *.nomeVariabileValido.get(). In caso positivo si testa se il nome trovato corrisponde effettivamente a un Item e si crea un evento di update nel ViewModel. Il compito diventa più difficile in caso di scorrimento di array di Item. In quest'eventualità si creano riferimenti a collegamenti possibili che determinano dei test nel ViewModel per gli eventi di update di ogni elemento che potrebbe influenzare il risultato della funzione. Discuteremo più avanti questi casi (par 4.3.1), evidenziando poi i motivi più significativi delle scelte effettuate (par 4.3.4).

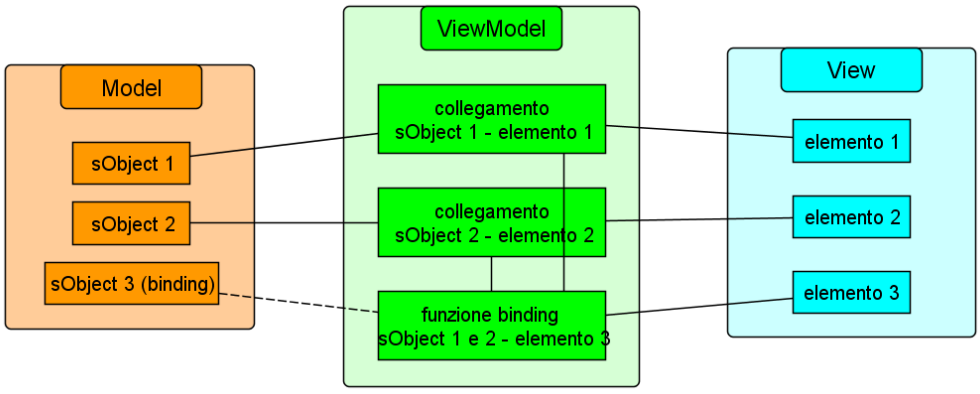

Figura 4.9: schema semplificato di binding tramite funzione

4.2.2 ItemSocket

Finora abbiamo considerato solo elementi in locale, ma nelle applicazioni web capita spesso di dover utilizzare fonti remote. Nel caso più banale, in cui i dati vengono caricati in locale e poi si può intervenire liberamente su di essi, non vengono forniti metodi ad hoc, poiché si ritiene che le tecniche AJAX siano sufficientemente complete [48]. Nel caso più interessante in cui i dati in remoto cambino con una certa frequenza, ed è quindi necessario garantire la consistenza tra loro e quelli utilizzati in locale, viene messo a disposizione l'oggetto ItemSocket. Questo si basa sugli WebSocket, introdotti dalle specifiche HTML5, che sono in grado di instaurare comunicazioni bidirezionali [49]. Questa possibilità è del tutto nuova poiché, sebbene ci fossero dei metodi per simulare una push dei dati da parte del server [17], non era previsto un oggetto standard per gestire adeguatamente la comunicazione tra le due parti.

ItemSocket è un oggetto atipico rispetto agli altri introdotti da easyHTML, poichè è prevista una controparte nel ViewModel ma non nella View. Il compito di un ItemSocket è quello di gestire gli aggiornamenti che vengono segnalati dal server, e fornire metodi per inviare eventualmente dei dati. ItemAdvanced e sObject hanno il metodo `attachSocket()` per poterli collegare tramite binding a un ItemSocket e quindi alla risorsa in remoto. L'argomento può essere l'ItemSocket, e in questo caso il collegamento è con tutto il messaggio ricevuto, oppure lo stesso seguito dell'elemento del messaggio JSON ricevuto dal server, il quale viene tradotto automaticamente e reso disponibile per l'utilizzo in JavaScript.

```
// Esempio di ItemSocket e collegamento a una risorsa
var remoto = new ItemSocket();
remoto.setHost("ws://resource_URL:port");
/* ipotizziamo che il messaggio ricevuto sia il seguente
{ "risultato": {"casa":0, "trasferta":1} } */
var partita_live = new Item();
partita_live.addString("casa", "trasferta");
partita_live.casa.attachSocket("remoto", "risultato.casa");
partita_live.trasferta.attachSocket("remoto",
"risultato.trasferta");
// a ogni aggiornamento da ws://resource_URL:port verrà
// aggiornato anche partita_live.casa e partita_live.trasferta
```

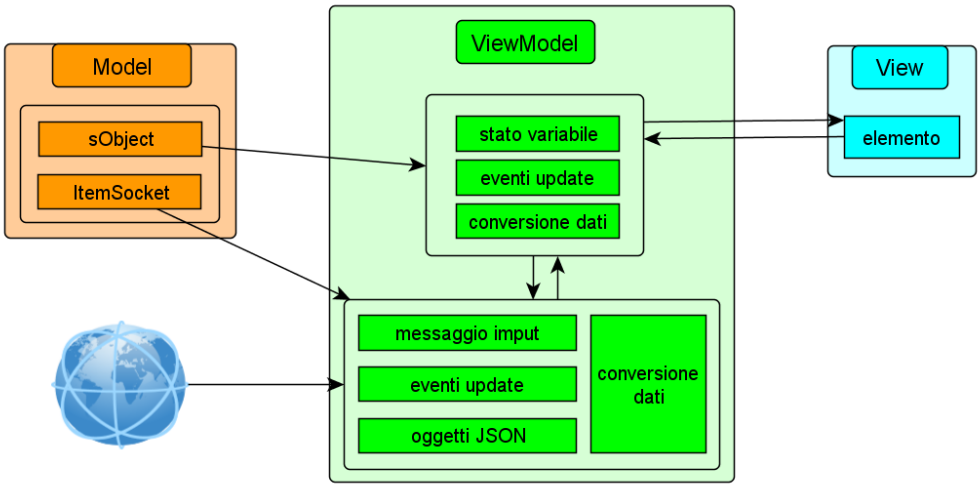

Figura 4.10: gestione dei binding a risorse in remoto

In figura 4.10 vediamo lo schema di gestione dei binding in remoto, dove si nota come lo stato delle variabili sia interamente localizzato nel ViewModel. A questo punto ci si potrebbe domandare se non fosse più sensato creare il gestore remoto all'interno dello stesso sObject, dal momento che l'ItemSocket non ha una sua rappresentazione nella View e che, dallo schema appena visto, appare quasi ridondante. Nelle applicazioni reali però è difficile che ci sia una singola informazione atomica che proviene da un server; appare più verosimile che un certo numero di dati siano residenti in remoto, e a questo punto creare molti oggetti WebSocket sarebbe inefficiente e poco sensato, poiché esso serve a instaurare una connessione con un server, dal quale poi possono essere inviati più dati contemporaneamente. Abbiamo preferito quindi realizzare un oggetto a sé stante, così da evitare l'overhead dovuto a troppi canali di comunicazione, e per riuscire a gestire l'eventuale messaggio JSON di risposta in un solo blocco. Del resto sarebbe stato più complicato anche l'invio di dati; difficile immaginare un server che gestisce ogni singola risorsa in modo autonomo, piuttosto che prevedere l'update più consistente dello stato locale fornito da una risposta completa.

4.2.3 Elementi HTML

Analizziamo più nel dettaglio gli elementi HTML che vanno a comporre la View. Abbiamo già visto quando tali elementi vengono generati (fig 4.1), ovvero dopo che il documento è stato completamente interpretato, me ora vediamo come avviene la disposizione effettiva.

Per prima cosa, mentre il documento HTML viene interpretato, ogni volta che è presente la funzione `here()` viene creato un elemento di tipo section al quale viene assegnato l'id presente nell'argomento, che deve corrispondere al nome di una variabile che identifica un Item o un ItemAdvanced. Nel caso delle funzioni `inside()` l'elemento viene creato allo stesso modo, ma inserito nella posizione indicata dal secondo argomento (può essere un Item o un id di un elemento qualunque del documento HTML). Questi contenitori vuoti verranno poi riempiti con altri elementi (derivanti dagli sObject per gli Item, mentre per gli ItemAdvanced dipendono dalla tipologia e dagli attributi), i quali andranno a costituire la View vera e propria.

Ogni volta che da un elemento del Model viene generato il corrispettivo nel ViewModel, la libreria utilizza gli attributi dell'oggetto in questione per stabilire quale sia la rappresentazione più opportuna nella View. Il tag da utilizzare dipende prima di tutto dal tipo di oggetto in questione (è un boolean o un numero? E' un elemento atomico o un elemento complesso come una mappa interattiva?), e successivamente dagli attributi e dallo stato (è collegato con un binding ad altre risorse? E' il locale o in remoto?).

Esponiamo nel dettaglio le scelte effettuate nella rappresentazione di un sNumber e di un ItemMap, mentre per visualizzare l'impatto visivo nella View di altri elementi si rimanda al paragrafo 5.3.

Un sNumber ha un certo numero di attributi da considerare per decidere quale sia l'elemento migliore per la sua rappresentazione nella View. Tra di essi troviamo `isInteger`, `min`, `max`, `modifiable`, `invisible`. Oltre a questi bisogna considerare se c'è un binding verso un altro sNumber, oppure un binding tramite funzione, o ancora un binding con una risorsa remota gestita da un ItemSocket. Negli ultimi 2 casi l'oggetto sarà visualizzato come un testo non modificabile, per la precisione l'elemento che conterrà il

valore in output è uno span. In caso di dati da una fonte in remoto sarà visualizzato un campo vuoto o il numero che rappresenta lo stato attuale della variabile, oppure un messaggio che indica l'impossibilità di recuperare il valore dal server.

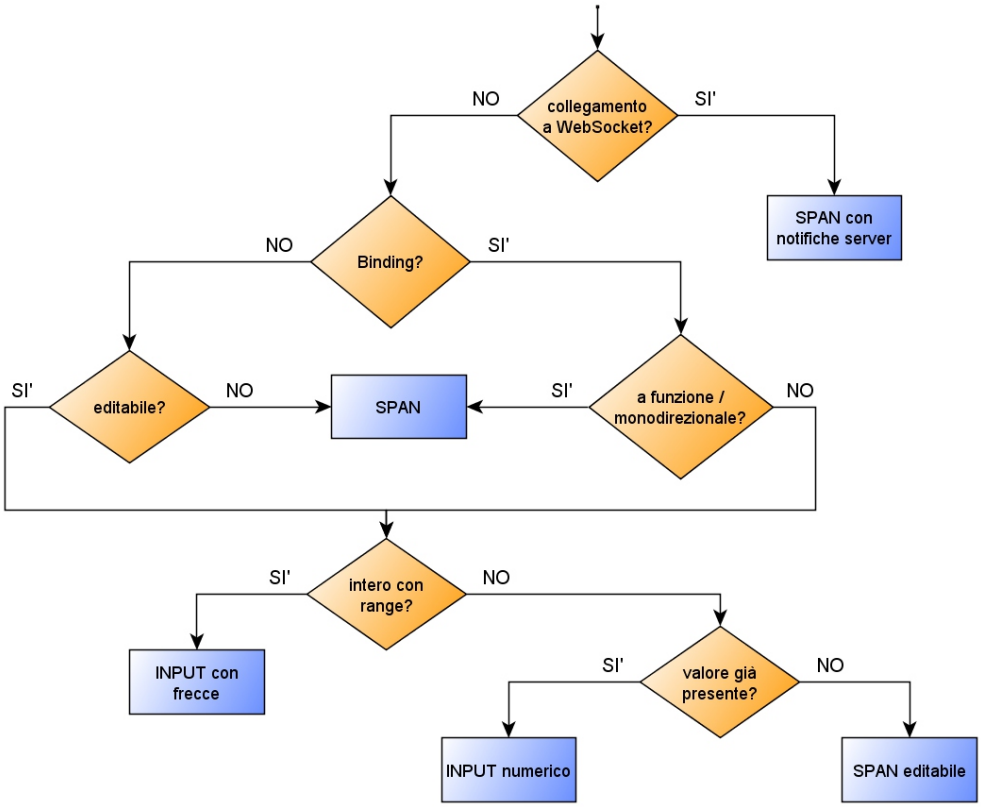

Figura 4.11: diagramma di scelta dell'elemento da creare per un sNumber

Nel caso in cui invece il valore sia intero e compreso entro limiti ridotti sarà visualizzato un input con le frecce dedicate all'incremento e decremento del valore. Se invece il numero fosse non intero oppure non sono definiti dei limiti, allora verrà controllato lo stato della variabile.

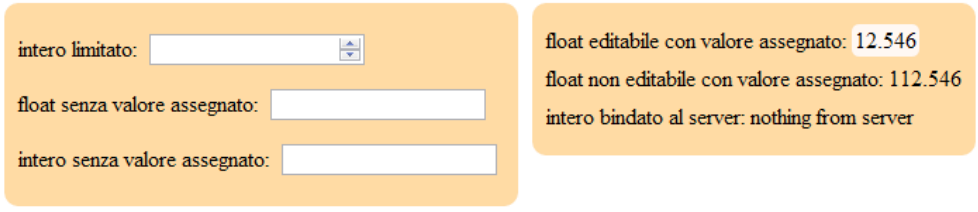

Figura 4.12: diversi elementi in base a stato della variabile e altri parametri

Se non è impostato alcun valore verrà creato un elemento di input che accetta solo la parte numerica, altrimenti si opterà per uno span contenente il testo. In quest'ultimo caso, se il valore attribuito alla variabile è modificabile, sarà possibile intervenire sul testo grazie all'attributo contenteditable posto a true. Per chiarire meglio le scelte possibili si veda il diagramma di flusso in figura 4.11 e le differenze rilevabili nell'interfaccia in figura 4.12.

Nel caso di ItemMap le scelte sono basate su parametri differenti. La prima discriminante sarà data dalla tipologia di visualizzazione, se normale o semplificata. Nel primo caso viene creato un iframe e viene poi composto l'attributo src da utilizzare per la creazione della mappa interattiva tramite Google Maps, mentre nel secondo caso l'output è testuale e le scelte sono più simili a quelle viste per l'sNumber. La seconda discriminante è data dall'utilizzo di coordinate o di un luogo significativo per indicare la posizione geografica. Infine bisogna controllare se si vuole indicare una singola località, un gruppo, un percorso. Nel caso di visualizzazione avanzata ci sono altri comandi da considerare come lo zoom e l'aggiunta di caselle di input per la modifica manuale dei parametri.

Tutti gli Item, ItemAdvanced e sObject seguono un percorso simile per determinare l'elemento (o gli elementi) da esporre nella View. Per esempi dei risultati visivi che si ottengono in situazioni diverse si rimanda al paragrafo 5.3.

4.2.4 Aspetti grafici

La composizione della View definisce gli elementi HTML, e determina le modalità d'interazione disponibili per l'utente finale. La parte grafica è invece da sviluppare tramite CSS, e viene realizzata in modo indipendente. Si può affermare che lo sviluppo dell'interfaccia passi attraverso la generazione della View e, in egual misura, la creazione di regole CSS, poiché l'impatto visivo è determinato soprattutto da quest'ultima parte (fig 4.9).

La libreria non si occupa di fornire metodi per la generazione delle regole d'impaginazione, ma lascia questo compito interamente allo sviluppatore, in

modo da consentire la maggiore flessibilità possibile (solitamente sono figure differenti a sviluppare la logica dell'applicazione e l'aspetto finale).

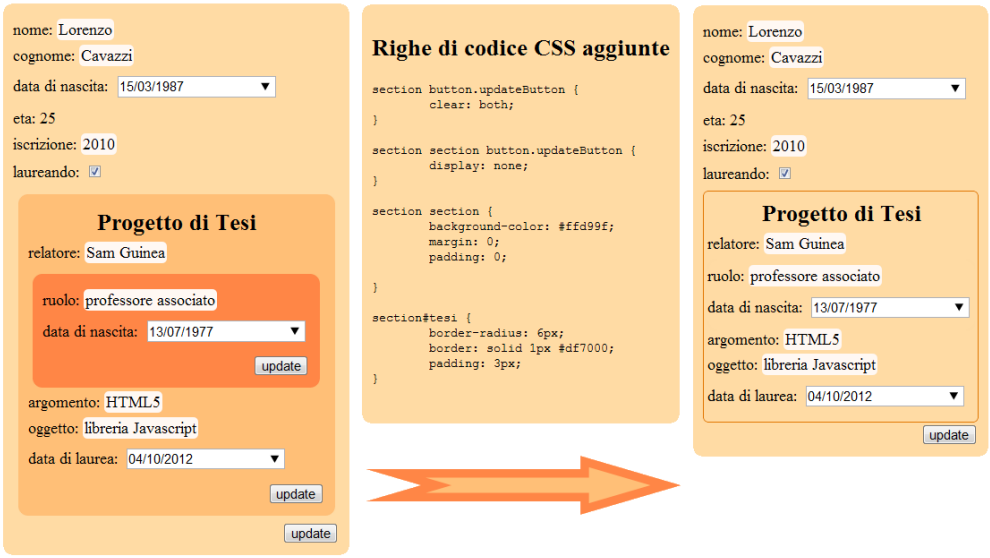

Figura 4.13: differenze nell'interfaccia determinate da diverse regole CSS

Poichè le regole CSS si basano sulla selezione degli elementi tramite DOM, e la stessa cosa viene fatta anche da un crescente numero di librerie come jQuery, si è cercato di assegnare automaticamente a ogni elemento generato degli id significativi e dei nomi di classe altrettanto validi per poter intervenire su tipologie d'elemento simili in modo immediato, senza dover definire classi custom. Viene inoltre data la possibilità di aggiungere nel Model degli attributi che verranno tramandati ai corrispondenti elementi HTML nella View, in modo da aumentare ancor di più le possibilità di personalizzazione.

Gli Item, che di fatto sono dei contenitori di sObject, avranno un elemento di tipo `section` come loro corrispondente nella View, più degli eventuali elementi testuali aggiuntivi. La `section` avrà id uguale al nome dell'Item, mentre gli elementi aggiuntivi avranno lo steso nome seguito da "_funzioneSvolta". Ad esempio l'etichetta che fa da titolo a un Item, se presente, verrà rappresentata come un h1 con id "nomeItem_label", mentre il tasto di update sarà "nomeItem_updateButton", e apparterrà alle classi "easyHTML_button" e "buttonUpdate". Per gli sObject e gli ItemAdvanced l'attribuzione degli id e delle classi è simile a quanto descritto per gli Item.

Nel caso in cui si utilizzino librerie che intervengo sui contenuti generati

nella View bisogna prestare attenzione a come questi si comportano in caso di aggiornamento dei dati. Gli elementi possono essere modificati, o addirittura eliminati e rimpiazzati da altri, eventualmente anche differenti. Si pensi ad esempio a un sNumber di tipo float a cui viene assegnato un valore dopo un update. Seguendo il diagramma di figura 4.11 notiamo che si passa da un input numerico a uno span editabile, quindi l'elemento nel DOM viene rimosso e sostituito da uno nuovo. A quel punto sarà necessario eseguire nuovamente il codice che apporta le modifiche. Si raccomanda quindi di porlo all'interno di una funzione da passare come argomento al metodo onUpdate degli elementi interessati, così da essere sicuri che, ogniqualvolta un evento modifichi gli elementi presenti, non venga azzerato l'effetto applicato.

4.3 Consistenza dei dati

Nel precedente paragrafo abbiamo introdotto le modalità di generazione del ViewModel e della View, evidenziando come il ViewModel si occupi di mantenere la consistenza dei dati. Tuttavia non siamo scesi sempre nei particolari, ma ci siamo limitati a parlare di eventi di update. In questa sezione approfondiremo i casi più interessanti, per i quali non è banale gestire la sincronizzazione fra i 3 componenti del pattern MVVM. Cercheremo inoltre di motivare le scelte fatte, specialmente nei casi dove un problema poteva essere affrontato in più modi differenti.

Si tenga sempre presente che tutta la fase di sincronia è affidata al ViewModel, poiché il Model e la View sono inconsapevoli l'uno dell'altra, mentre il primo fa da collante tra di essi andando a leggere i dati da entrambi e gestendo tutti gli eventi che vengono scatenati.

4.3.1 Eventi dal Model o dalla View

La maggior parte degli eventi provengono originariamente dal Model o dalla View, ma gli effetti che determinano si avvertono nel ViewModel. Chiariamo meglio quest'affermazione con un esempio. Supponiamo che l'utente intervenga nell'interfaccia aggiornando un valore, ad esempio lo sNumber `età` contenuto in un Item che rappresenta una persona. L'evento viene generato dalla View, e viene propagato nel ViewModel. A questo punto l'evento originale è terminato, ma nel ViewModel possono essere lanciati altri eventi. Se il valore immesso non è valido, non c'è nessuna propagazione; se invece assume un valore potenzialmente corretto, il ViewModel genera una serie di eventi che possono coinvolgere potenzialmente tutte le parti di tutti i 3 componenti. Se lo sNumber non aveva ancora un valore assegnato, per prima cosa viene influenzato l'elemento della View che ha scatenato l'evento, modificandolo da input a un'altra tipologia (fig 4.11). Poi è possibile che venga

propagato il valore al Model (non è detto, dipende dalle impostazioni; è possibile non tenere traccia nel Model dei cambiamenti dello stato delle variabili a runtime, salvandoli solo nel ViewModel così da garantire la persistenza fino a che l'applicazione viene chiusa ma non oltre). A questo punto la catena di eventi potrebbe continuare, e ciò dipende il grafo delle dipendenze. Nel ViewModel viene generato un grafo delle dipendenze quando, per tutti gli elementi del Model, è stato creato un corrispettivo controllore. Quest'albero tiene conto dei binding tra ItemSocket, ItemAdvanced e Item (per gli sObject si considera la dipendenza tra i rispettivi Item padri), e viene consultato ogni volta che ci sono variazioni nello stato delle variabili.

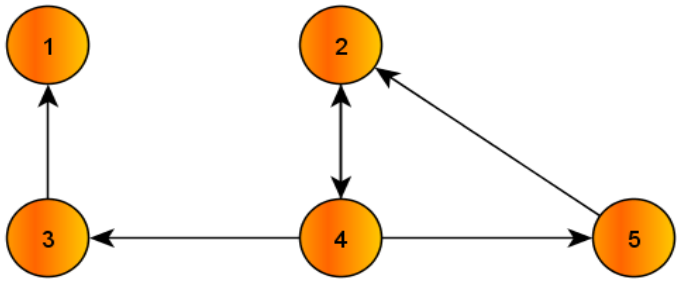

Figura 4.14: esempio di possibile deadlock. Un aggiornamento allo stato della risorsa 1 genera un loop tra 2-4-5

Se questi dovessero avere effetti su altri elementi, l'evento viene propagato a quelli che subiscono effettivamente delle modifiche, i quali vengono gestiti scatenando una nuova catena di update che riguarda le loro controparti View e Model e di nuovo le eventuali dipendenze. Prima di lanciare effettivamente gli eventi, viene fatto un controllo per stabilire quali parti sono potenzialmente coinvolte, eliminando ogni volta i nodi già esplorati così da evitare la generazione di loop infiniti (fig 4.14).

Quando invece una modifica viene effettuata allo stato di una variabile del Model, un evento viene propagato e gestito allo stesso modo degli eventi della View, ma solo nel caso in cui l'applicazione sia a runtime. Questa discriminazione è necessaria perchè eventuali interventi sul Model effettuati prima che il documento sia stato completamente interpretato non sono modifiche, ma semplicemente le fasi di creazione del Model stesso

(ricordiamo che, ogni volta che l'applicazione viene ricaricata, il codice viene letto nuovamente e reinterpretato). Questo distinguo è superfluo per la View poiché, essendo generata dalla libreria, non esiste fin quando l'applicazione non è a runtime.

Garantire la sincronia diventa più difficoltoso nel caso di scorrimento di array di Item all'interno di una funzione di binding. Supponiamo che l'Item `fattura` contenga lo sObject `totale` che è dato dalla somma di tutti i prezzi nell'array di Item `acquisto`. Il codice che scorre l'array sarà simile al seguente:

```
fattura.totale.setBinding(function(){
	var totale = 0;
	for (var i = 0; i < acquisti.length; i++) {
		totale += acquisti[i].valore.get();
	}
	return totale;
});
```

Ma come determinare in questo caso il grafo delle dipendenze? Fintanto che il codice non viene eseguito non è possibile sapere con certezza a quali Item si accederà scorrendo l'array. Se ci fosse un costrutto `if` posto all'interno del ciclo `for` che esegue un controllo per esclude dal calcolo alcuni elementi, non potremmo determinare quali di essi verrebbero scelti solo guardando il codice. Per risolvere questo problema l'albero generato prevede alcune dipendenze certe e altre possibili. In questo caso `fattura` presenta una dipendenza possibile con tutti gli Item nell'array `acquisti`, e perciò verrà effettuato un controllo sul possibile cambiamento dello stato della variabile `totale` ogni volta che ci sarà un cambiamento potenzialmente in grado di influenzare il valore ritornato dal binding, sebbene ciò potrebbe non accadere.

Questi controlli aggiuntivi sono necessari, poiché un array come quello visto nell'esempio potrebbe vedere l'aggiunta o la rimozione di altri Item in fase di runtime, rendendo a questo punto del tutto impossibile assicurare la consistenza dei dati se si lavorasse solo con dipendenze certe (par 4.3.4).

4.3.2 LocalStorage

L'utilizzo del LocalStorage rappresenta il metodo più semplice per rendere persistenti i cambiamenti effettuati a runtime. E' possibile decidere se mantenere l'ultimo stato valido delle variabili per ogni Item e ItemAdvanced utilizzando il metodo `setLocalStorage(true/false)`. Impostando il valore a `true`, al prossimo avvio dell'applicazione, i dati in locale vengono recuperati e vanno a sovrascrivere i contenuti di base.

Poichè la memorizzazione in locale sovrascrive i dati che provengono dal Model, quando un oggetto ha l'attributo `localStorage` impostato a `true` viene generato nella View anche un pulsante di reset per poter ripristinare i valori di default, andando in quel caso a sovrascrivere sia lo stato nel ViewModel (e quindi degli altri componenti a cui viene propagato dal gestore di eventi), sia i valori presenti nel LocalStorage. Ovviamente è possibile anche evitare l'aggiunta del tasto reset e invocare tale evento via codice, in modo da consentire un controllo personalizzato da parte del programmatore, oppure disabilitare del tutto i salvataggi in locale a runtime, rendendo le successive modifiche solo temporanee (si pensi alla necessità di effettuare dei test senza cancellare i dati precedentemente memorizzati).

Il LocalStorage può essere utilizzato anche per mantenere la consistenza dei dati fra pagine diverse della stessa applicazione. Si pensi ad esempio a un'applicazione che visualizza un form per l'inserimento dei dati, rappresentato da un Item con una serie di sObject vuoti. Dopo la validazione dei dati si viene rediretti verso un'altra pagina, in cui vengono presentati altri Item, ma allo stesso tempo è necessario utilizzare i dati inseriti in precedenza. Se il dominio resta lo stesso, tutto ciò che è contenuto nel LocalStorage resta accessibile, perciò è sufficiente creare un Item con lo stesso nome e le stesse tipologie di dato per poter utilizzare di nuovo i valori inseriti in precedenza.

Si tenga presente che questo approccio non garantisce la consistenza fra più pagine web contemporaneamente attive. I dati vengono infatti caricati solo alla fine dell'interpretazione del documento, dopo di che lo stato delle variabili utilizzato è quello nel ViewModel. In caso di aggiornamento ci sarà un

salvataggio verso il LocalStorage, ma fino alla chiusura dell'applicazione non ci saranno più caricamenti da esso. Lo stato dei dati di un'altra pagina non verrebbe quindi influenzato in nessun modo, poiché lo stato locale si riferisce a quanto presente nel proprio ViewModel. Bisogna quindi prestare attenzione nell'utilizzo del LocalStorage, che può essere usato per “trasportare” i dati fra più pagine, ma non per mantenere la sincronia fra di esse a runtime.

4.3.3 Dati asincroni

Abbiamo già visto che l'oggetto ItemSocket viene utilizzato per importare i dati da remoto sfruttando gli WebSocket introdotti con HTML5 (par 4.2.2). Approfondiamo ora il meccanismo che consente di mantenere la consistenza dei dati con la View, e distinguiamo i casi in cui è possibile utilizzare le funzionalità della libreria da quelli in cui conviene importare i dati dall'esterno con altri metodi come AJAX.

Per prima cosa sottolineiamo che, in caso di dati da remoto gestiti tramite ItemSocket, lo stato attuale delle variabili ricevuto non viene mai salvato permanentemente in locale. Gli WebSocket sono pensati come canale di comunicazione bidirezionale, per cui sono particolarmente indicati in casi in cui il server abbia necessità di inviare spesso notifiche di cambiamento dello stato. Ha senso utilizzare questo strumento se effettivamente è necessario un aggiornamento in tempo reale da parte dal server. Si pensi ad esempio a un Item che visualizza il risultato di un evento sportivo; il punteggio, il minuto attuale, la lista degli ammoniti e una serie di altri valori subisce cambiamenti continui, per cui è sensato appoggiarsi ad un ItemSocket.

Quest'oggetto, una volta a runtime, instaura una connessione utilizzando un link specificato dal metodo `setURL()`, e scatena un evento ogni volta che riceve dei nuovi messaggi, il viene gestito nel ViewModel andando a stimolare tutti gli oggetti collegati all'ItemSocket, che recuperano i nuovi dati e li visualizzano. Se il messaggio ricevuto è in formato JSON, esso viene tradotto automaticamente in oggetti JavaScript ai quali è possibile accede direttamente

dagli sObject utilizzando il metodo `attachSocket(itemSocket, object.property)`, dove `object.property` rappresenta la proprietà dell'oggetto ricevuto dal server a cui si vuole accedere. Viene garantita l'integrità di eventuali sObject collegati tramite binding (sia diretti che tramite funzione) a un oggetto qualsiasi che abbia come fonte un ItemSocket, sfruttando la propagazione a catena degli eventi (par 4.2.1).

In molte situazione i dati provenienti da remoto sono utilizzati per costruire il Model, o per assegnare agli oggetti un valore iniziale. Questo scenario è tipico della maggior parte delle applicazioni web attuali, poiché strumenti come IndexedDB e WebSQL, introdotti anch'essi da HTML5, sono ancora poco o per nulla utilizzati, e i dati vengono memorizzati per lo più all'interno di database SQL, dai quali è necessario estrarre i dati lato server. In questo caso la logica di caricamento dati si avvale solitamente di chiamate AJAX, che sono già piuttosto semplici e si prestano poco ad automatizzazione (non posso prevedere i pochi parametri da passare come argomento, quindi un oggetto per la loro gestione sarebbe superfluo). Si lascia quindi allo sviluppatore l'incarico di gestire questo scambio dei dati in modo autonomo, richiamando nuovamente il metodo `onUpdate()` come strumento per definire il codice che gestisce l'eventuale salvataggio in remoto delle modifiche locali agli Item e ItemAdvanced.

Proponiamo ora un esempio in cui tutti i dati sono in remoto, ma alcuni di esse sono statici e quindi vengono caricati con una chiamata AJAX al server, mentre altri sono altamente dinamici e gestiti dall'oggetto ItemSocket.

```
// recupero dati dal server con chiamata AJAX
var data;
$.ajax({ type: "POST", url: "anURL.php", data: {options:
type_in}}).done(function(msg) {
      data = msg;
});

// uso i dati per creare il Model
var partita = new Item();
partita.addString("campionato", "scasa",
"strasferta").addNumber("giornata").addDate("data");
partita.setValue({campionato:data.campionato, scasa:data.scasa})
// ...
partita.setLabel({scasa: "squadra in casa", strasferta: "squadra in
trasferta"});
```

```
// creo websocket
var sockItem = new ItemSocket();
sockItem.setHost("ws://anotherURL");

// aggiungo i binding al socket
partita.addNumber("goal_casa", "goal_trasferta");
partita.goal_casa.attachSocket("sockItem", "goal_casa");
partita.goal_trasferta.attachSocket("sockItem", "goal_trasferta");
```

4.3.4 Modifiche dinamiche

Abbiamo visto i meccanismi utilizzati dalla libreria per mantenere la consistenza dei dati per ogni tipologia di binding. Tuttavia ci siamo finora limitati ad analizzare la gestione dei cambiamenti dello stato delle variabili che contengono valori significativi. In un'applicazione fortemente dinamica potrebbe però essere necessario creare Item e ItemAdvanced a runtime, e aggiornare di conseguenza ViewModel, View e grafo delle dipendenze.

EasyHTML mette a disposizione una serie di funzioni per creare un elemento del Model e aggiungerlo dinamicamente nella View a runtime, aggiornando l'interfaccia e tutti i componenti. Quest'operazione può essere piuttosto gravosa nel caso in cui vengano definiti binding tramite funzione che coinvolgono un elevato numero di oggetti, poiché l'aggiornamento del grafo delle dipendenze richiede lo scorrimento di tutti elementi per aggiornare gli eventi di update. Tuttavia il più delle volte il nuovo oggetto è un elemento posizionato in un array dove sono già presenti altri Item (si pensi all'aggiunta di un oggetto al carrello, o all'inserimento di un nuovo studente nell'array degli iscritti alla facoltà d'ingegneria), e che quindi non viene referenziato singolarmente da qualche sObject, ma coinvolto all'interno di cicli `while` o `for`. Non risulta in tal caso necessario aggiornare il grafo delle dipendenze poiché, per lo scorrimento di array, la dipendenza possibile creata nell'albero è già rivolta verso tutti gli elementi facenti parte dell'array stesso, indipendentemente da quando vengono creati. Se invece l'elemento presenta molte dipendenze aggiuntive, la libreria potrebbe impiegare più tempo per

l'elaborazione in base alle risorse hardware disponibili per il browser.

```
// esempio di aggiunta a runtime di Item
var destinazioni = [];
var meta = new Item();
meta.addString("destinazione").addNumber("giorni").setLabel("tappa")

function aggiungi() {
	var last = destinazioni.length;
	destinazioni.push(new Item("meta"));
	H5.runtime.attachItemInsideId("x", "destinazioni["+last+"]");
}

<a href="#" onclick="aggiungi();">aggiungi tappa</a>
```

Anche la modifica agli attributi strutturali del Model è richiesto un aggiornamento del ViewModel e della View nella maggior parte dei casi. Ad esempio, se rendo un sObject non modificabile, è necessario cambiare le modalità d'interazione possibili nell'interfaccia utente, sostituendo eventualmente l'elemento con un altro più adatto. Se invece modifico un attributo come la tipologia in un sNumber, passando da integer a float, sarà necessario intervenire sul ViewModel così da rivedere le funzioni per la validazione dei dati. Tutto questo viene fatto invocando la funzione `redraw()` che, se eseguita a runtime, attua tutte le modifiche agli oggetti.

CAPITOLO 5

VALIDAZIONE

Validare una libreria JavaScript non è un compito facile, in particolare quando si cerca di creare uno strumento che vuole esplorare vie alternative per risolvere problemi per cui esistono già diversi approcci. Andiamo quindi a esprimere sia un confronto con gli obiettivi che ci siamo posti, sia con gli strumenti già presenti nello stato dell'arte. In quest'ultimo caso dobbiamo comunque tener presenti le finalità per cui questo lavoro è stato svolto; alcune caratteristiche possono risultare migliori per determinate applicazioni ma penalizzanti per altre, per cui cerchiamo di operare le dovute discriminazioni.

Un elemento importante per la validazione è costituito dal materiale messo a disposizione dello sviluppatore. L'ostacolo principale nell'adottare una libreria, in particolar modo se creata ex novo e con una sintassi del tutto propria, riguarda la difficoltà nel comprendere in breve tempo come utilizzare le funzioni messe a disposizione. Deve essere chiaro fin da subito cosa è possibile fare e quali sono i punti di forza (e quelli deboli), poiché richiedere di consultare tutta la reference per decidere se easyHTML può soddisfare delle specifiche esigenze è quantomeno pretenzioso.

La realizzazione di un numero adeguato di tutorial interattivi risulta parte

integrante del lavoro svolto, tanto che questi sono stati aggiornati ad ogni aggiunta di funzionalità e proposti a degli sviluppatori per capire man mano la bontà del lavoro svolto. Ciò ha consentito da un lato di validare la libreria con l'aiuto di un gruppo di programmatori HTML (che rappresentano il target finale dell'applicazione), e dall'altro di creare un ambiente interattivo dove provare a modificare il codice per ottenere in tempo reale l'output che genera la libreria.

Infine proponiamo un esempio di applicazione completa realizzata con easyHTML, sia per dimostrare le possibilità della libreria, sia per fornire un lavoro che possa servire da spunto agli sviluppatori.

5.1 Confronti e valutazioni

Proponiamo in questa sezione delle valutazioni sulla bontà del lavoro fatto basandoci su 3 parametri di giudizio differenti. Per primo ci confrontiamo con gli obiettivi che ci siamo posti nel paragrafo 3.1, per verificare quali abbiamo effettivamente raggiunto e quali invece sono stati solo parzialmente centrati a causa delle difficoltà sorte durante lo sviluppo. Poi vediamo un confronto con quanto offre lo stato dell'arte, così da evidenziare i punti forti della libreria, ovvero i contesti in cui è proficuo utilizzare le funzionalità messe a disposizione. Per finire facciamo qualche considerazione in merito alle prestazioni che si possono ottenere. In questo modo è possibile valutare la complessità gestibile dalla libreria (e cogliamo anche l'occasione per motivare qualche scelta effettuata durante l'implementazione).

5.1.1 Valutazione degli obiettivi raggiunti

Nel paragrafo 3.1 abbiamo definito quattro obiettivi principali e ora, dopo aver descritto a fondo la libreria, vediamo se possiamo considerarli raggiunti.

Il punto di partenza del lavoro svolto è la consistenza dei dati, per cui l'obiettivo più importante è quello di mantenere sempre sincronizzati i dati utilizzati nella logica applicativa con quelli esposti nell'interfaccia utente. La libreria prevede la creazione del ViewModel a partire dai dati del Model, e di generare degli eventi per segnalare eventuali interventi su quest'ultimo a runtime. Inoltre la definizione della View è quasi completamente automatizzata e anche in questo caso delle funzioni hanno il compito di trasmettere al livello sottostante le interazioni avvenute. La propagazione dell'informazione tra i diversi livelli non rappresenta quindi un problema, ma la parte più delicata riguarda le dipendenze. Nei paragrafi 4.2.1 e 4.2.2 abbiamo visto tutti i possibili binding, inclusi quelli definiti dall'utente. Essi rappresentano il caso più delicato poiché non possiamo avere pieno controllo sul codice di una

funzione definita dal programmatore. Tuttavia abbiamo visto come eventuali influenze di uno o più oggetti verso un altro siano memorizzare in un grafo creato dalla libreria insieme al ViewModel, ed eventualmente aggiornato nel caso di modifiche strutturali a runtime (par 4.3.4). In questo modo siamo certi che, all'interno di tale componente, le notifiche siano tramandate anche per via orizzontale, così da stimolare tutti i cambiamenti a catena generati da un'interazione sia nel Model che nella View.

Il secondo obiettivo è inerente la flessibilità dell'interfaccia. Lo scopo è quello di poter modificare dinamicamente la tipologia di elemento esposta per rappresentare un oggetto, così da gestire il cambio d'interazione a disposizione dell'utente con le modalità standard previste da HTML5 (ovvero cambio di tag). Per fare ciò abbiamo deciso di generare automaticamente la View, dando allo sviluppatore la possibilità di definire solo alcuni dettagli (par 4.2.3). Il risultato si può dire raggiunto, tuttavia in questo modo impediamo un pieno controllo sulle modalità con cui avviene l'output a video (e parzialmente anche sull'input). Un'idea emersa per risolvere il problema prevedeva di aggiungere la possibilità di indicare un qualsiasi tipo di elemento da associare a un oggetto del Model; purtroppo l'ipotesi è stata scartata poiché richiede una complessità di gestione troppo elevata, col rischio di rendere la libreria poco immediata e troppo statica la View (se lo sviluppatore indica un elemento da utilizzare allora non c'è più dinamicità, e corriamo il rischio di allinearci con la soluzione proposta da Knockout, ma con una sintassi addirittura più complessa).

Il terzo obiettivo consiste nello sgravare lo sviluppatore da compiti ripetitivi e facilmente automatizzabili. Abbiamo descritto ampiamente le parti generate in automatico dalla libreria e fornito esempi di codice con rispettivi risultati che dimostrano l'immediatezza delle soluzioni a disposizione. Possiamo considerare questo come il punto forte di easyHTML, tanto da aver scelto il nome della libreria proprio per indicare la semplicità con cui è possibile creare un oggetto che garantisca la consistenza (par 4.1.1).

Infine il quarto obiettivo riguarda l'utilità e l'appetibilità della libreria. Stabilire se e quanto questo punto sia stato centrato è difficile, poiché le considerazioni da fare variano molto a seconda della tipologia di applicazione da sviluppare, del target specifico a cui ci riferiamo e della confidenza

nell'utilizzo della sintassi tipica di un linguaggio object oriented (non è da dare per scontata per un sviluppatore HTML5 che utilizza principalmente JavaScript). Ci auguriamo di aver aumentato le possibilità di raggiungere questo punto con l'aggiunta di strumenti come i tutorial interattivi, i siti dimostrativi e la documentazione dettagliata consultabile sul sito di riferimento. Per considerazioni più pratiche su questo punto rimandiamo invece ai prossimi 2 paragrafi.

Riassumiamo schematicamente le osservazioni fatte

OBIETTIVI	STATO
Mantenere la consistenza dei dati tra logica applicativa e interfaccia	PIENAMENTE RAGGIUNTO
Garantire flessibilità all'interfaccia in base a contesto e stato delle variabili	PARZIALMENTE RAGGIUNTO
Sgravare lo sviluppatore da compiti ripetitivi e facilmente automatizzabili	PIENAMENTE RAGGIUNTO
Rendere la libreria uno strumento realmente utile e appetibile	PARZIALMENTE RAGGIUNTO

5.1.2 Confronto con lo stato dell'arte

Il termine di paragone utilizzato più spesso per confrontare easyHTML con lo stato dell'arte è la libreria Knockout, che rappresenta la concorrente più vicina sia per le funzionalità offerte che come tipologia di applicazione che permette di sviluppare. Cerchiamo di evidenziare le differenze tra le soluzioni offerte dalle 2 in alcuni casi significativi.

Riportiamo qui sotto il codice necessario per creare un oggetto nel Model che rappresenta uno studente e la sua controparte nella View. Ci limitiamo ad inserire alcuni dati di base e a creare un binding con un secondo oggetto che identifica un professore, in modo da determinare qualche dipendenza nel ViewModel. In figura 5.1 riportiamo anche l'output che si ottiene a video

eseguendo il codice in un browser.

```
// CODICE Knockout

// Model e ViewModel
function Studente(nome, cognome, eta, laureando, relatore) {
	var self = this;
	self.nome = ko.observable(nome);
	self.cognome = ko.observable(cognome);
	self.eta = ko.observable(eta);
	self.laureando = ko.observable(laureando);
	self.relatore = ko.computed(relatore);
}
function Professore(nome, cognome) {
	var self = this;
	self.nome = ko.observable(nome);
	self.cognome = ko.observable(cognome);
}
function ViewModel() {
	var self = this;
	self.professore = new Professore("Sam", "Guinea");
	self.studente = new Studente("Lorenzo", "Cavazzi", 24, true,
	    function(){
	return self.professore.nome() +" "+ self.professore.cognome();
	});
}
ko.applyBindings(new ViewModel());

// View
<section>
	<li>
		<label>nome: </label>
		<input data-bind="value: studente.nome"  />
	</li>
	<li>
		<label>cognome: </label>
		<input data-bind="value: studente.cognome"  />
	</li>
	<li>
		<label>eta: </label>
		<input data-bind="value: studente.eta"  />
	</li>
	<li>
		<label>laureando: </label>
		<input type="checkbox" data-bind="checked:
studente.laureando"  />
	</li>
	<li>
		<span>relatore: </span>
		<span data-bind="text: studente.relatore"></span>
	</li>
</section>
```

```
// CODICE easyHTML

// Model e ViewModel
var studente = new Item();
studente.addString("nome:Lorenzo", "cognome:Cavazzi").addNumber
("eta:24").addBoolean("laureando:true").addString("relatore");
var professore = new Item();
professore.addString("nome:Sam", "cognome:Guinea");
studente.relatore.setBinding(function(){
      return professore.nome.get() +" "+ professore.cognome.get();
});

// View
<script>here("studente");</script>
```

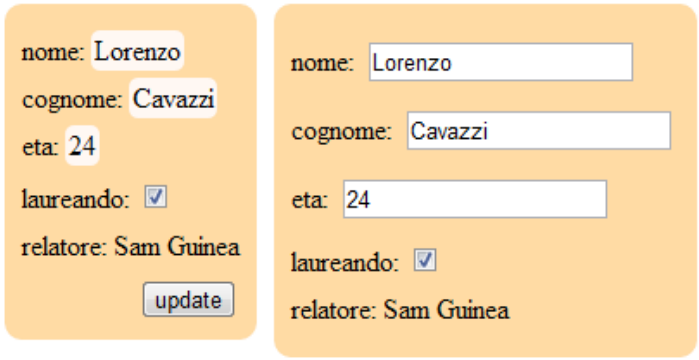

Figura 5.1: esempio di output del codice easyHTML (sinistra) e Knockout

Il primo effetto che salta all'occhio è la differenza nella quantità di codice prodotto per ottenere risultati simili tra la soluzione che utilizza Knockout e quella con easyHTML. Nel secondo caso non è stato creato un oggetto che facesse da classe, ma viene utilizzata la sintassi abbreviata per creare e definire i valori di un elemento. Anche volendo prima definire separatamente la classe sarebbe stato necessario aggiungere solo altre 2 righe di codice (tuttavia `studente` e `professore` possono essere utilizzati come modelli per definire un altro elemento con gli stessi attributi, per cui di fatto tale modifica sarebbe ridondante in ogni caso).

La creazione del Model non differisce di molto, e nemmeno determinare dipendenze personalizzate all'interno del ViewModel (si noti come la sintassi per definire `studente.relatore` sia praticamente identica). Tuttavia il drastico calo nel numero di righe prodotte si deve alla diversa implementazione della View che in un caso deve essere esplicitata, mentre

nell'altro risulta totalmente automatizzata. Con Knockout non solo bisogna creare tutti gli elementi, ma è necessario anche assegnare manualmente i binding alle risorse della View con gli attributi `data-bind`. Per tutti i campi si rendere necessario scegliere un determinato elemento per visualizzarli e permettere all'utente d'interagire: ad esempio per `nome` e `cognome` si è scelto un elemento `input`, ma sarebbe più appropriato utilizzarlo solo quando il campo è vuoto, e passare a uno `span` editabile quando alla variabile viene assegnato un valore. Questo non è possibile se non con la definizione di funzioni piuttosto intricate per gestire la rimozione e l'aggiunta di elementi a tunrime. Al contrario con easyHTML un comportamento delle genere è quello previsto di base.

Vediamo ora qualche considerazione in merito alla semplicità nello scorrimento di array di elementi.

```
// Knockout (fa porre dentro la funzione ViewModel)
self.studenti = ko.observableArray();
var tmpStudente = new Studente("nome", "cognome", null, null,
function(){
	// codice funzione
});
self.studenti.push(tmpStudente);
```

```
// easyHTML
var studenti = [];
var tmpStudente = new Item("studente");
tmpStudente.setValue(nome: "nome", cognome; "cognome");
tmpStudente.relatore.setBinding(function(){
	// codice funzione
})
studenti.push(tmpStudente);
```

La creazione di un array non differisce di molto, anche se con Knockout sono costretto a creare tale oggetto all'interno della funzione adibita alla gestione del ViewModel. Anche la differenza tra un observableArray e la sua controparte standard non è rilevante, sebbene utilizzare una sintassi custom è sempre un piccolo ostacolo. Non viene riportata tutta la parte di codice inerente la creazione e definizione di nuovi elementi, nemmeno quella per gestire tale situazione a runtime (sarebbe troppo lunga e poco significativa), mentre approfondiamo lo scorrimento dei valori memorizzati.

```
// Knockout
self.statistiche = {
	"laureandi": ko.computed(function(){
		var tot = 0;
		ko.utils.arrayForEach(self.studenti, function(stud){
			if (stud.laureand() === true)
				tot++
		});
		return tot;
	})
}
```

```
// easyHTML
var statistiche = new Item();
statistiche.addNumber("laureando");
statistiche.laureandi.setBinding(function(){
	for (var n=0, var tot=0; n<statistiche.length; n++) {
		if (studente[n].laureando.get() === true)
			tot++;
	}
	return tot;
});
```

Il numero di righe di codice è anche in questo caso paragonabile, tuttavia la sintassi utilizzata in Knockout è decisamente meno intuitiva. Lo scorrimento dell'array è affidato a una funzione creata ad hoc dalla libreria, e il suo impiego non è naturale per un programmatore, facendo sì che debba prima imparare il corretto modo per implementarla. Riteniamo che l'approccio di easyHTML sia decisamente più semplice da memorizzare, e questo è sicuramente un grande punto di forza.

Un ulteriore vantaggio riguarda la creazione di applicazioni che coinvolgono un numero significativo di oggetti. Con easyHTML la logica di business e i binding tramite funzione sono distribuiti per ogni Item, mentre in Knockout sono concentrati in un'unica funzione che tende a crescere velocemente. Riteniamo che il nostro approccio consenta di mantenere più facilmente il codice nel caso di espansioni e modifiche da effettuare durante il ciclo vita del software.

Quello che è stato presentato come un punto di forza, ovvero la flessibilità della View, potrebbe rappresentare in alcuni casi particolari un vincolo alla personalizzazione. Per esempio un programmatore potrebbe voler esporre modalità d'interazione non standard nella View, e questo non sarebbe

possibile con easyHTML. Anche con Knockout c'è da sottolineare che si può gestire i contenuti solo di alcuni elementi, tuttavia è più semplice trovare soluzioni creative fintanto che si spazia fra di essi. In particolare è possibile gestire un elevato numero di eventi dell'interfaccia piuttosto che limitarsi alla modifica di elementi come nella soluzione che abbiamo proposto. Sicuramente questo rappresenta un limite per easyHTML, sul quale contiamo d'intervenire in futuro.

Un altro vantaggio di Knockout è quello di poter lavorare molto agevolmente nel caso in cui si voglia aggiungere e rimuovere elementi a runtime. Le funzioni disponibili sono versatili anche per la nostra libreria, ma il loro utilizzo è meno immediato. Vediamo un esempio di codice per dare la possibilità all'utente di aggiungere o eliminare elementi del Model tramite interfaccia.

```
// Knockout
self.aggiungi = function() {
	self.studenti.push(new Studente(/* init code */));
}
self.rimuovi = function() {
	self.tappe.pop();
}
```

```
// easyHTML
function aggiungi() {
	var last = tappe.length;
	tappe.push(new Item("tappa"));
	H5.runtime.attachItemInsideId("quiDestinazioni",
"tappe["+last+"]");
	tappe[last].setValue(/* init code */);
}
function rimuovi() {
	var last = tappe.length - 1;
	if (last >= 0) {
		$('#tappe['+last+']').remove();
	}
	tappe.pop();
}
```

In Knockout la gestione della View per gli array di elementi viene definita sfruttando dei binding particolari, così da generare solo una volta il l'elemento (o il gruppo di elementi) da inserire nel documento, e la libreria gestisce eventuali aggiunte e rimozioni. Non è quindi necessario intervenire sulla View

come nel caso di easyHTML, dove bisogna creare e rimuovere esplicitamente ogni elemento in aggiunta o da togliere.

Vediamo una tabella che ricapitola quanto fin qui osservato.

	Knockout	easyHTML
velocità stesura codice		
dinamicità elementi della View		
personalizzazione binding		
scorrimento array		
gestione di un numero elevato di elementi		
flessibilità tipologie di elemento		
gestione degli eventi		
aggiunta e rimozione elementi a runtime		

5.1.3 Prestazioni

Nell'introdurre la programmazione in HTML5 abbiamo più volte sottolineato come la portabilità del codice sia un enorme vantaggio rispetto ad altri linguaggi. Quando si parla di programmi particolarmente avidi di risorse è probabile che, a prescindere dalla piattaforma usata per lo sviluppo, ci si concentri verso uno specifico sistema operativo, o comunque si facciano ottimizzazioni dedicate all'ambiente d'esecuzione. EasyHTML non è indirizzata verso la progettazione di applicazioni di questo tipo, molto esigenti dal punto di vista dell'hardware, poiché ci aspettiamo che per esse vengano scelte soluzioni maggiormente personalizzate.

Per applicazioni più leggere, con le quali l'utilizzo della libreria garantisce l'immediatezza di realizzazione per la quale è stata concepita, ci si aspetta di potervi accedere anche da dispositivi portatili come smartphone e tablet, tipicamente dotati di risorse hardware limitate. Poichè il contesto d'esecuzione del codice della struttura MVVM è il terminale utilizzato dall'utente, qualche considerazione sulle prestazioni si rende necessaria.

Per prima cosa proponiamo un paragone tra easyHTML e Knockout nell'esecuzione del codice necessario per visualizzare l'output di figura 5.1 (fig 5.2). Lo strumento utilizzato per la verifica è JavaScript Performance playground (jsPerf [50]), un punto di riferimento per il confronto di codice in questo linguaggio [51]. La piattaforma d'esecuzione del codice è un laptop abbastanza datato equipaggiato con Windows 7 e Google Chrome 21.

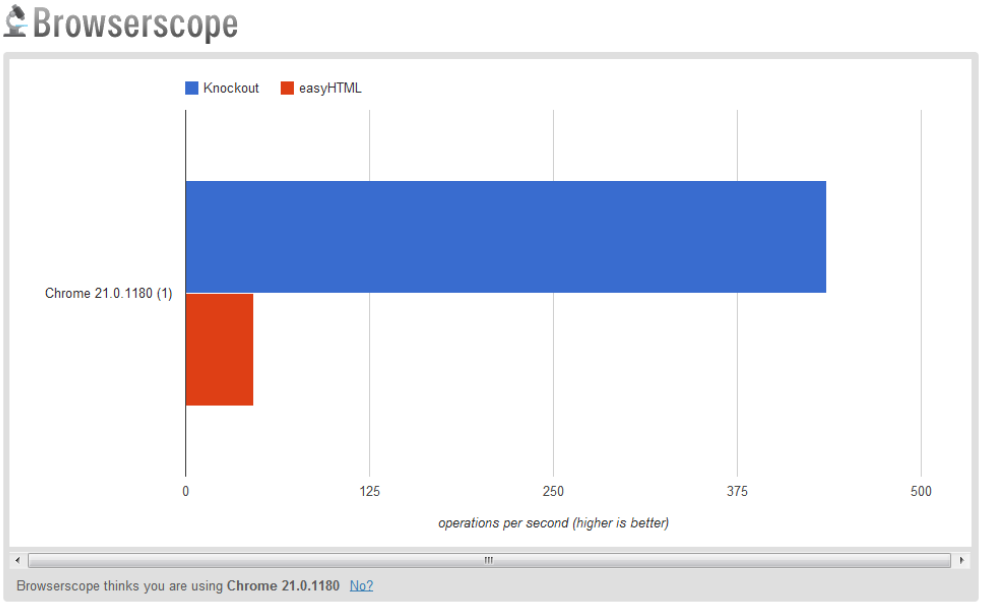

Figura 5.2: confronto fra prestazioni di Knockout e easyHTML

Come si può notare Knockout garantisce delle prestazioni decisamente migliori, circa un ordine di grandezza a parità di risultato che si vuole ottenere. Tuttavia bisogna considerare che easyHTML spende la maggior parte del tempo per scorrere gli elementi dell'oggetto `window` per identificare tutti gli Item e gli ItemAdvanced, così da costruire un grafo completo delle dipendenze. Questo implica che, al crescere del numero di elementi, la complessità aumenti in proporzione minore a quella diretta.

In figura 5.3 vediamo il grafico delle performance ottenuto per un'applicazione molto più complessa, che prevede un elevato numero di dipendenze e un maggior numero di Item e ItemAdvanced. L'output che si ottiene è quello di figura 5.4 e il primo caso di test prevede di generare solo gli oggetti visualizzati in figura, il secondo la stessa situazione ma con 20 elementi in più nella struttura iniziale.

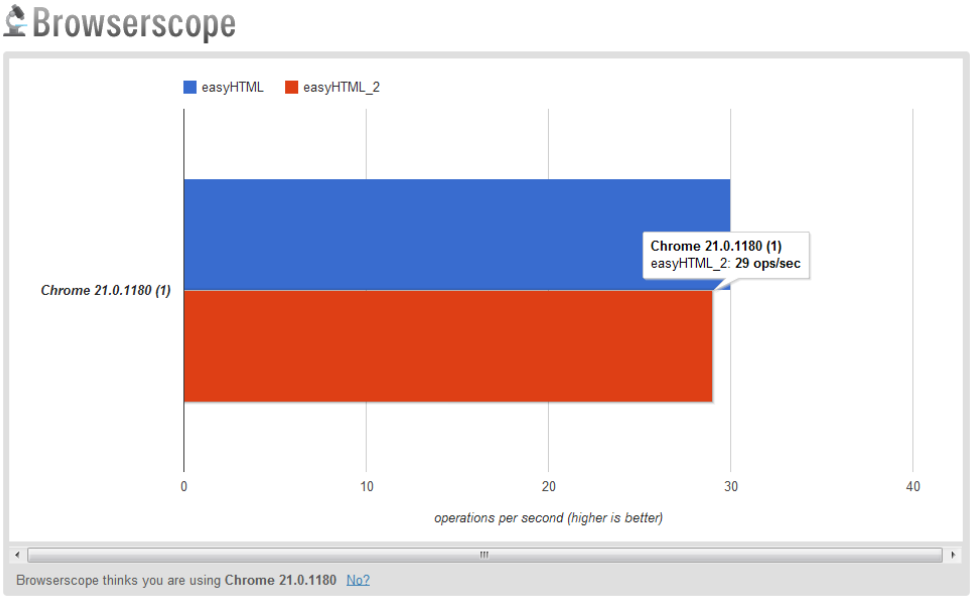

Figura 5.3: prestazioni di easyHTML con molti elementi

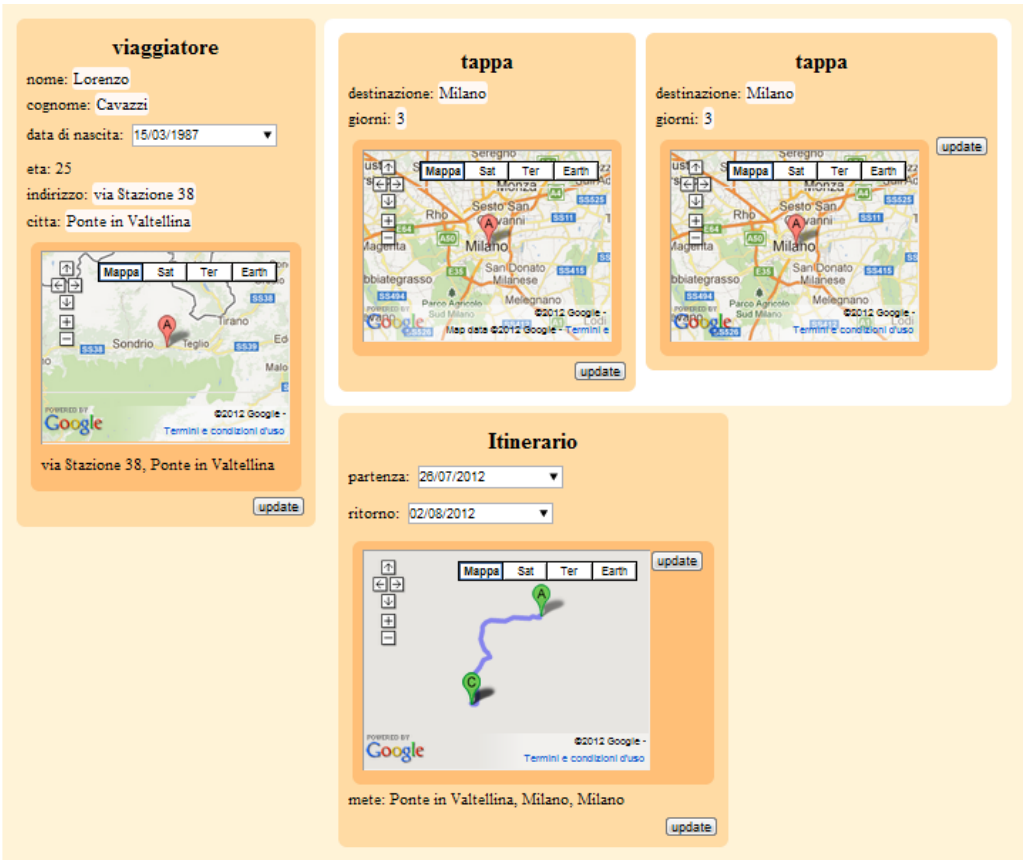

Figura 5.4: output ottenuto con codice d'esempio usato per i test di figura 5.3

Nel test case ho 4 Item e 4 ItemMap e un numero cospicuo di binding definiti tramite funzione (non si riporta il codice per esteso poiché sarebbe troppo lungo). La misura delle performance riporta un valore nel primo caso di 30op/s, mentre nel secondo (con l'aggiunta di altri 10 Item e 10 ItemMap) di 29op/s. La differenza è minima poiché l'influenza sul grafo delle dipendenze è quasi nulla tra i due contesti. Ci sono più elementi da gestire, ma questi determinano solo dipendenze possibili già rilevate in precedenza, per cui la libreria non deve compiere lavoro aggiuntivo (par 4.3.4).

Un confronto con Knockout in questo caso non è possibile, poiché non ci sono metodi per gestire le mappe (sebbene esistano plugin sperimentali per implementarle questa funzionalità [52]). Ma ci interessa di più fare un paragone coi risultati ottenuto in precedenza. Le operazioni al secondo passano da circa 45 a 30, che corrisponde a un peggioramento del 33% delle performance. Questo però a fronte di una complessità della logica molto più elevata, e ciò fa capire come la libreria sia in grado di scalare bene per applicazioni di media complessità.

Un valore del genere corrisponde a un tempo di circa 35ms per la gestione di tutte le componenti MVVM. Anche considerando casi peggiori è difficile superare il valore limite di 100ms, che è più che accettabile considerando le tempistiche necessarie per caricare il documento e interpretarlo, normalmente nell'ordine di grandezza dei 100-1000ms. Troviamo conferma di quanto detto in figura 5.5 dove viene riportato il tempo di caricamento segnalato dalla console di Google Chrome. Tutte le risorse vengono caricate in 250ms e la libreria svolge ancora lavoro per un tempo di circa 25ms (si veda la distanza tra la barra arancione della quarta riga e la linea blu, che definisce la durata appena indicata).

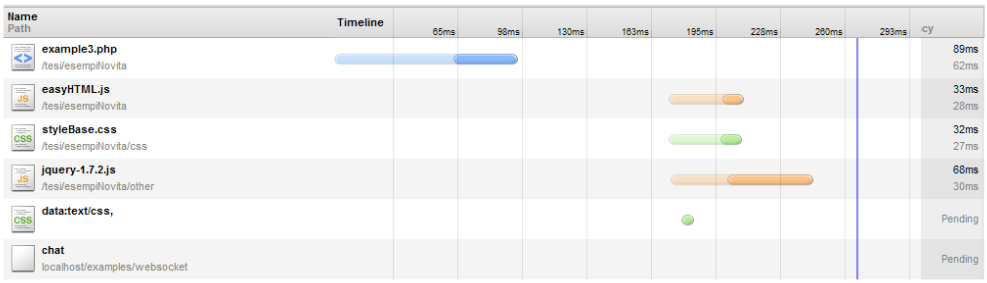

Figura 5.5: tempistiche per caricare le risorse di una pagina HTML

Purtroppo non ci è possibile svolgere i medesimi test per piattaforme come android e iOs, poiché jsPerf non le supporta a pieno (o meglio dovrebbe supportarle, ma vengono generati degli errori sull'evento `window.onload` che non sono imputabili alla libreria ma al browser, e ciò accade anche con Knockout in alcune situazioni). Ci limitiamo a osservare che, in tali ambienti, le tempistiche tendono ad aumentare di circa 2-3 volte, ma riteniamo il risultato ancora accettabile poiché rimane comunque inferiore al tempo di caricamento e interpretazione delle altre risorse del documento.

Per quanto riguarda le modifiche a runtime, esse occupano un tempo sicuramente inferiore a quello appena visto. Per quanto profondi siano gli interventi da effettuare su ViewModel e View a fronte di un'interazione, il caso pessimo prevede la ridefinizione di ogni elemento tra quelli già coinvolti nelle dipendenze, che rispetto alle operazioni effettuate dalla libreria alla fine dell'interpretazione del documento aggiunge solo la rimozione di elementi (le tempistiche sono, in proporzione, trascurabili). Un valore <100ms è da ritenersi più che accettabile, sebbene lontano dagli standard di Knockout.

5.2 Strumenti per lo sviluppatore

Creare una libreria valida e semplice da usare sarebbe un lavoro inutile senza un'adeguata documentazione e senza dei mezzi per dimostrarne le potenzialità e facilitare l'apprendimento della sintassi. Gli sviluppatori interessati devono aver modo di apprendere in fretta come utilizzare la libreria e magari avere uno strumento per testarla all'interno di qualche contesto d'esempio tipico.

Per questi motivi abbiamo creato un sito con tutta la documentazione necessaria per cominciare a utilizzare easyHTML in autonomia. Ci sono dei tutorial interattivi per spiegare passo passo come iniziare a creare un'applicazione che si basa sull'architettura MVVM, e delle pagine con esempi più completi dai quali prendere spunto per elaborare soluzioni che rispondano alle esigenze più complesse. Di seguito presentiamo brevemente quello che abbiamo realizzato, ma invitiamo il lettore più interessato a visitare il sito di riferimento www.cavazziweb.it/easyhtml.php

5.2.1 Documentazione

La pagina di documentazione (fig 5.6) riporta in alto un link alla reference, disponibile con o senza descrizione. Nel primo caso vengono illustrati dettagliatamente tutti gli oggetti creati dalla libreria e i metodi definiti su di essi, mentre il documento più compatto riporta solo un elenco selle funzioni disponibili senza esempi. Si fa notare che le stesse descrizioni sono riportate sotto forma di codice ScriptDoc all'interno della libreria, e sono pertanto consultabili anche durante la programmazione, a patto che si utilizzi un'IDE che supporti questo tipo di documentazione.

Successivamente c'è un elenco espandibile di tutorial, che approfondiamo nel prossimo paragrafo. Abbiamo pensato di creare anche degli esempi di codice di confronto con Knockout e Backbone, in modo da permettere ai

programmatori che già utilizzano queste librerie di verificare al volo le differenze con delle applicazioni simili tra loro.

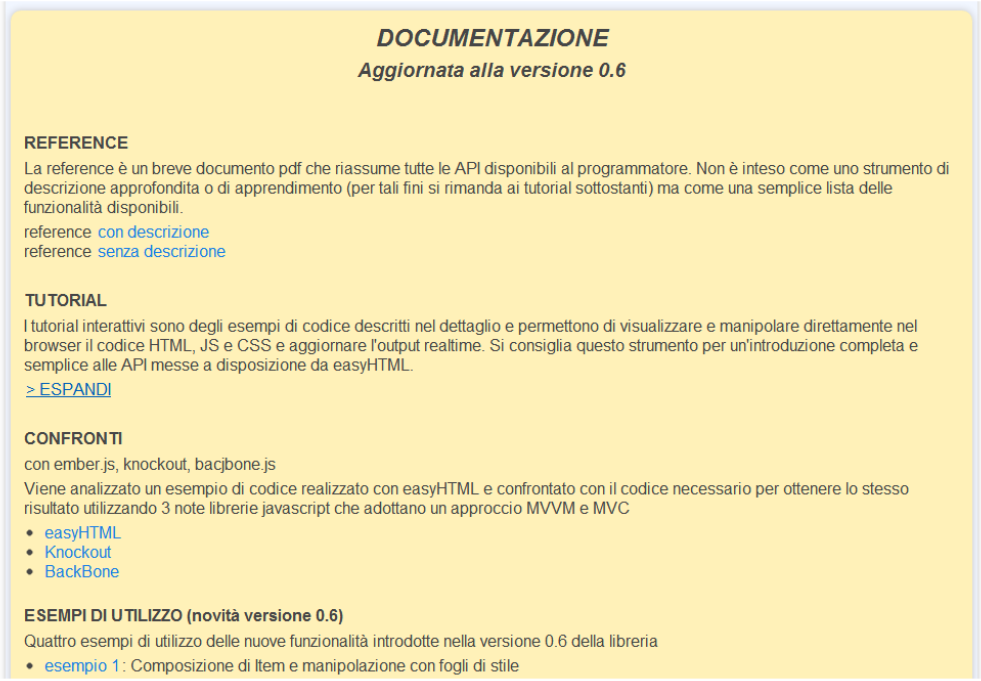

Figura 5.6: schermata della documentazione sul sito

Infine troviamo degli esempi più completi in cui si sfruttano a pieno le potenzialità della libreria. I primi si concentrano sulle novità introdotte nelle ultime versioni (in particolare la gestione degli ItemAdvanced e ItemSocket, che potrebbero creare qualche difficoltà in più rispetto agli Item). Gli ultimi sono decisamente più corposi e rappresentano dei potenziali siti che si basano interamente su easyHTML (par 5.3).

5.2.2 Tutorial

I tutorial sono esempi interattivi in cui l'utente può modificare direttamente nel browser il codice HTML e JavaScript e visualizzare in tempo reale i risultati che si ottengono. Rappresentano il modo più veloce per imparare a usare la libreria, e sono impostati in modo da guidare il programmatore attraverso un

percorso di apprendimento graduale.

I primi blocchi introducono gli Item e gli sObject, per poi presentare i metodi disponibili per le loro istanze. Poi si passa ai binding semplici e quelli tramite funzione, spiegando come poterne creare di nuovi anche a runtime. Infine si descrivono gli ItemAdvanced e gli ItemSocket, mostrando qualche esempio e proponendo la creazione di alcune parti di codice.

Figura 5.7: esempio di tutorial interattivo

In figura 5.7 riportiamo l'esempio di un tutorial così come appare online (leggermente riadattato per questioni d'impaginazione). A sinistra c'è il codice HTML di base, a destra invece il JavaScript opportunamente commentato e con una serie di istruzioni per effettuare cambiamenti e provare a implementare piccole parti aggiuntive. In basso appare l'output determinato dal codice sorgente, e viene aggiornato a ogni cambiamento da parte dell'utente.

5.3 Casi d'uso

Abbiamo creato dei casi d'uso più completi rispetto ai tutorial per dare la possibilità agli sviluppatori di vedere alcune soluzioni utilizzate per risolvere problemi non banali. Presentiamo brevemente quelli che riteniamo più significativi, estrapolando delle parti di codice e proponendo l'output a video (lo stile applicato, pur non essendo fra i più piacevoli, ci serve a evidenziare alcuni dettagli sulle modalità d'interazione).

Un elemento che potrebbe creare qualche problema è ItemSocket. Supponiamo di voler seguire un evento in diretta, nello specifico una partita di calcio. Ovviamente abbiamo bisogno di un server che supporti il push di dati, e immaginiamo che ciò avvenga con messaggi JSON.

```
// Creazione oggetto ItemSocket
var sockItem = new ItemSocket();
sockItem.setHost("ws://localhost:8080/examples/websocket/chat");

// Gestione logica ItemSocket
squadra[0].goal.attachSocket("sockItem", "goalCasa");
squadra[1].goal.attachSocket("sockItem", "goalTrasferta");
live.minuto.attachSocket("sockItem", "minuto");
live.risultato.setBinding( function(){
	if (squadra[0].goal.get() !== undefined) {
		return squadra[0].goal.get() + " - " +
                                        squadra[1].goal.get();
	}
	else {
		// [...]
	}
});
```

Nel codice presentato l'ItemSocket viene istanziato e collegato alla risorsa in remoto. Non serve intervenire in altro modo poiché la gestione della ricezione dei messaggi è automatizzata, come anche le eventuali conversioni di dato. Per utilizzare le risposte dobbiamo effettuare un binding tra elementi già creati nel Model e l'ItemSocket. In caso di messaggi in formato JSON possiamo passare il nome della variabile target come secondo argomento della funzione `attachSocket()`, utilizzando per esso la sintassi JavaScript prevista per un oggetto qualunque.

A questo punto si sono creare delle dipendenze con l'ItemSocket, e

possiamo dare per scontata la consistenza dei dati a ogni update da parte del server. Volendo possiamo creare una funzione di binding che utilizza quei dati in modo indiretto, cioè passando dall'elemento per cui abbiamo appena definito il collegamento. La catena di eventi si occuperà di replicare le modifiche ove necessario (come avviene per `live.risultato`).

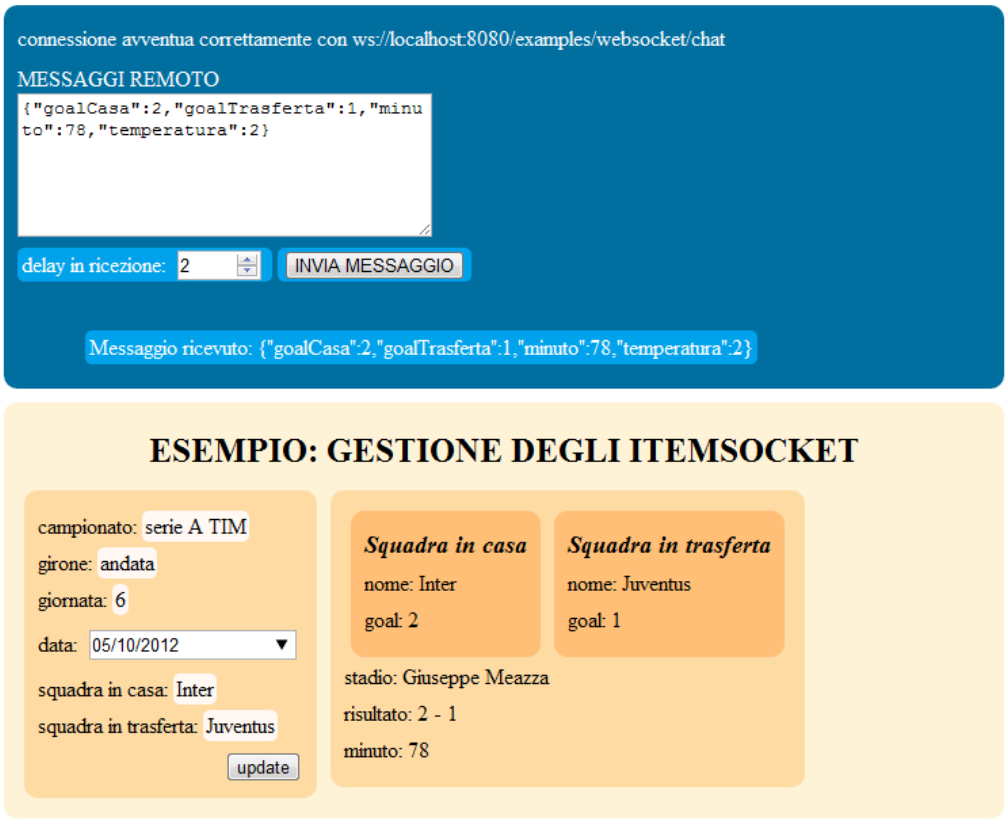

Figura 5.8: esempio completo di utilizzo di un ItemSocket

In figura 5.8 vediamo il messaggio ricevuto e l'output a video. Facciamo notare l'impossibilità d'intervenire sugli elementi presi da remoto; non è consentito modificare lo stato, e l'unico modo per aggirare l'ostacolo prevede di specificare la possibilità d'inviare eventuali update (per ora non supportata in automatico da easyHTML, andrebbe creata una funzione ad hoc che utilizza la funzione `myItemSocket.send(message)`).

Un altro caso interessante riguarda l'aggiunta di elementi interattivi a runtime, con conseguente aggiornamento degli Item e ItemAdvanced già presenti. Questa volta immaginiamo di voler determinare le tappe di un viaggio fornendo all'utente la possibilità di poter aggiungere o togliere delle

fermate intermedie, definirne la durata, visualizzare una mappa interattiva e aggiornare a ogni modifica un Item riassuntivo e un ItemMap indicante il percorso effettuato. Concentriamoci sul codice di gestione delle modifiche a runtime.

```
// gestione modifiche a runtime
var destinazioni = [];
var destinazioniMappe = [];
var meta = new Item();
meta.addString("destinazione:Milano").addNumber("giorni:3")
                                                    .setLabel("tappa");
function aggiungi() {
      var last = destinazioni.length;
      destinazioni.push(new Item("meta"));
      destinazioniMappe.push(new ItemMap);
      destinazioniMappe[last].setZoom(9).setDimension
                                        (260, 200).hideUpdate(true);
      destinazioniMappe[last].setBinding("destinazioni["+last+"]
                                                .destinazione");
      H5.runtime.attachItemInsideId("quiDestinazioni",
                                        "destinazioni["+last+"]");
      H5.after("destinazioniMappe["+last+"]", "giorni");
      H5.runtime.attachItemMapInsideItem("destinazioniMappe["
                              +last+"]", "destinazioni["+last+"]");
}

function togli() {
      var last = destinazioni.length - 1;
      if (last >= 0) {
            H5.runtime.remove('#destinazioni['+last+']');
            H5.runtime.remove('#destinazioniMappe['+last+']');
            destinazioni.pop();
            destinazioniMappe.pop();
      }
}
```

In questo caso dobbiamo creare delle funzioni JavaScript da richiamare quando viene premuto un pulsante `aggiungi` o `rimuovi`. Al loro interno dobbiamo gestire la logica di aggiunta o rimozione dell'elemento nel Model e nella View. le funzioni `H5.runtime.x` garantiscono l'update del ViewModel mentre si interviene sulla View. Finora abbiamo sempre visto esempi che fanno uso di `here()` o `inside()`, ma esse non garantirebbero il corretto comportamento degli elementi poiché non andrebbero ad aggiornare il grafo delle dipendenze. Si noti nella funzione `togli()` la necessità d'intervenire separatamente sul Model, rimuovendo anche qui l'oggetto. Bisogna prestare

attenzione a questo particolare poiché la libreria non può intervenire in automatico anche qui (magari il programmatore vuole mantenere l'Item nel Model ma eliminarlo dalla View, per cui la rimozione in automatico da parte di entrambe gli elementi sarebbe insensata).

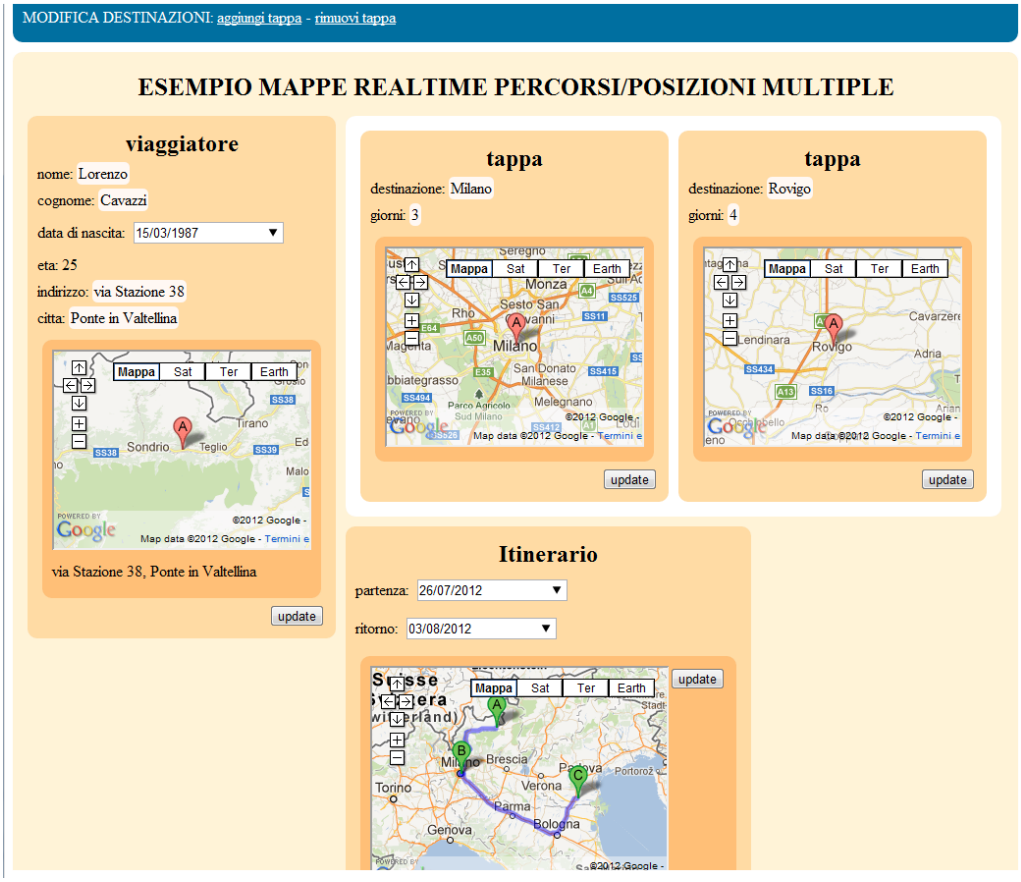

Figura 5.9: esempio di mappe con binding aggiornati a runtime

Nell'immagine 5.9 si distinguono 4 Item con associati degli ItemMap. L'ultimo blocco di elementi sfrutta le informazioni dei primi tre per aggiornare la mappa interattiva col percorso effettuato e la data di arrivo, la quale viene determinata in base alla partenza e ai giorni di sosta per ogni tappa (come conferma del mantenimento della consistenza dei dati in caso di modifiche si faccia un confronto con fig 5.4 dove si utilizza lo stesso esempio ma con lo stato di alcune variabili differente).

5.3.1 GiraMondo

GiraMondo è un'applicazione completa basata su easyHTML che consente agli utenti di registrarsi e tenere traccia dei propri viaggi. E' una specie di social network dove si possono aggiungere foto, video e altre informazioni da condividere con gli amici. E' un esempio completo per un programmatore (il codice sorgente è commentato in modo approfondito), e in particolare permette di trarre spunto da alcune soluzioni utilizzate per aumentare l'interattività (ad esempio la creazione di eventi non standard) e gestire gli elementi della View di easyHTML con librerie di terze parti (come gli effetti applicabili con jQuery).

Purtroppo possiamo solo limitarci a riportare qualche screenshot, ma invitiamo il lettore interessato a visionare il risultato completo al link cavazziweb.it/giramondo.

Figura 5.10: pagina dei viaggi

In figura 5.10 proponiamo la pagina dei viaggi. L'estrazione dal database

avviene con una chiamata Ajax che preleva i dati di tutti i viaggi che rientrano nei criteri di filtraggio della colonna di destra. In base alle opzioni attive si possono cambiare le interazioni e i dati visualizzati. Una volta ricevuta la risposta dal server gli elementi vengono creati come un array di Item e visualizzati.

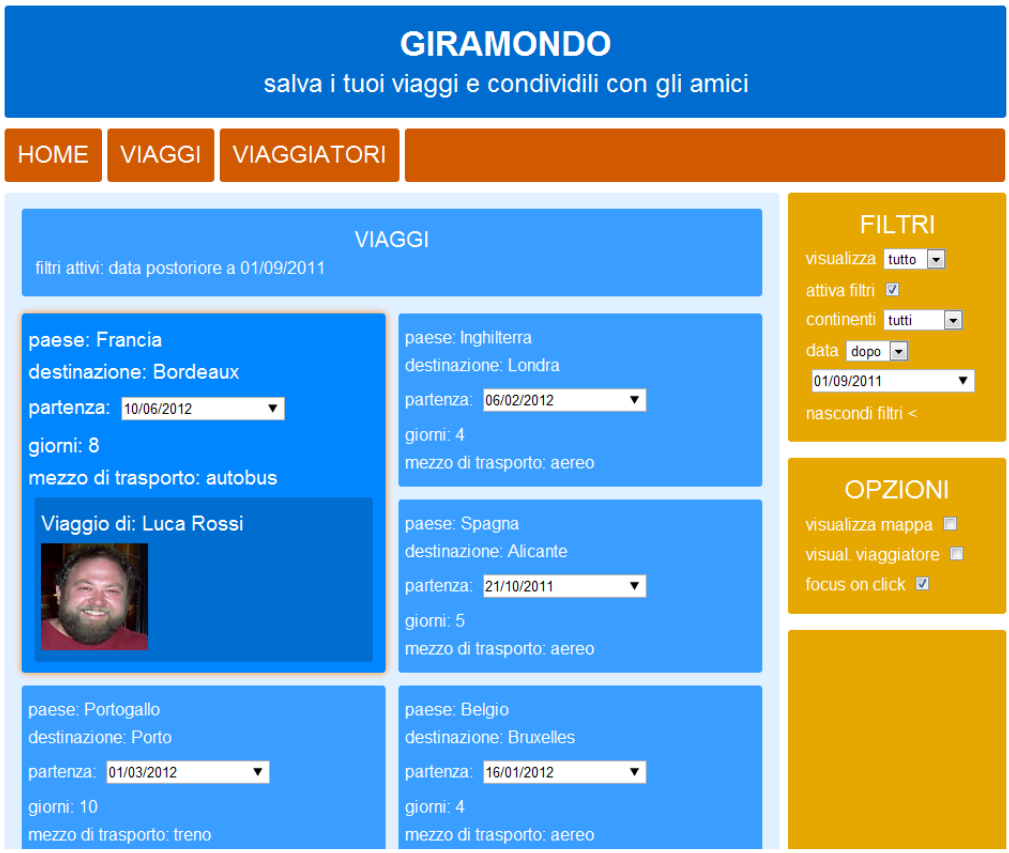

Figura 5.11: interazioni create con librerie di terze parti

In figura 5.11 vediamo un esempio di interazione aggiunta con una libreria di terze parti. In particolare abbiamo usato jQuery per catturare l'evento click su una section e stimolare il recupero di dati aggiuntivi dal server (il nome del viaggiatore e la sua foto profilo rimpicciolita) per visualizzarli con un Item e un ItemImage all'interno dell'Item già presente. Gli effetti visivi riguardano l'aumento della dimensione del carattere, la bordatura e lo sfondo.

In figura 5.12 vediamo il profilo di un utente loggato e le interazioni che può compiere. Dal menu di destra di nota che l'opzione di abilitazione delle modifiche è spuntata. Non ci sono pulsanti visibili per non appesantire la grafica, ma posizionandosi su un elemento è possibile modificarne il

contenuto e dare l'update col tasto che nel frattempo sarà comparso.

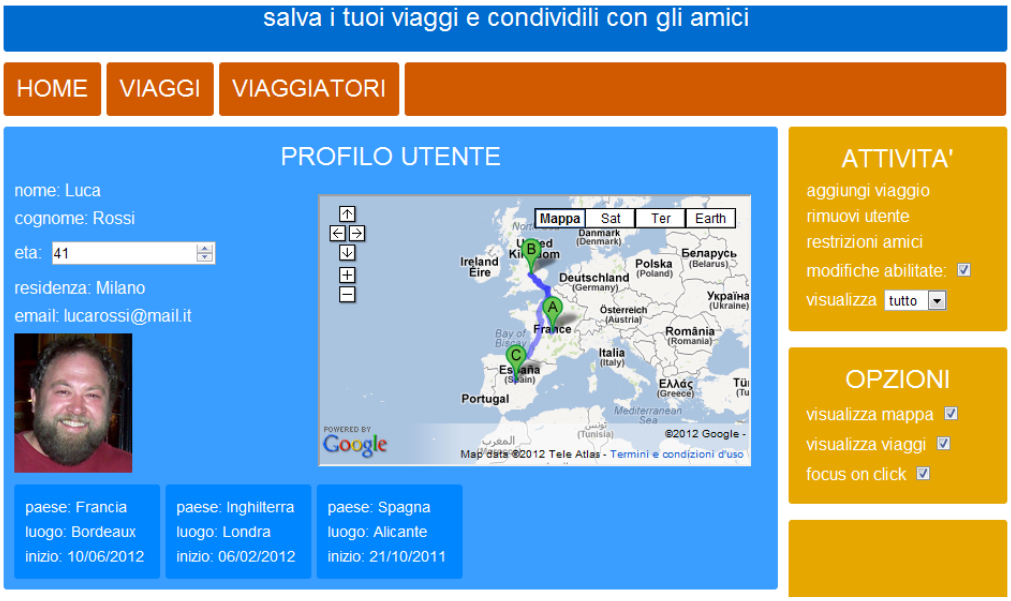

Figura 5.12: profilo di un utente loggato

In figura 5.13 vediamo cosa accade provando ad effettuare una modifica veloce a un viaggio. Cliccando sull'elemento c'è un impatto grafico differente (testo più grande, bordo evidenziato, tonalità dei colori differenti) e la comparsa a video di altre opzioni. La data (modificabile) viene proposta con un elemento di input, uno sfondo indica in modo più chiaro la possibilità di

Figura 5.13: effetti sulla View di una modifica al profilo

modifica del contenuto e compare in basso a destrail tasto update per fissare nel server i cambiamenti effettuati. Anche in questo caso si fa uso di una chiamata AJAX poiché sarebbe poco significativo l'uso di un ItemSocket (la comunicazione è una tantum da parte del client, inutile instaurare un canale permanente per scambiare messaggi).

CAPITOLO 6

CONCLUSIONI

Siamo giunti al termine della presentazione del lavoro svolto. Abbiamo approfondito il contesto nel quale ci siamo mossi, dandone un'inquadratura generale e sottolineando i problemi principiali ai quali abbiamo provato a porre rimedio. L'attenzione è stata rivolta ad alcuni punti specifici piuttosto che ad altri (par 3.1) nel tentativo di proporre un'idea veramente innovativa e che potesse avere un seguito al di fuori dell'ambito accademico.

Ci rendiamo conto che lo strumento proposto presenta ancora delle incompletezze, e l'intenzione è quella di migliorarlo nel prossimo futuro. Proponiamo ora alcune delle idee che stiamo discutendo per rendere easyHTML una libreria più completa e per cercare di renderla uno strumento ancora più semplice e immediato, andando magari a soddisfare le esigenze di un target più inesperto o che abbia la necessità di creare applicazioni “from scratch” in breve tempo (par 6.1.1). Esploriamo anche l'ipotesi ben più ambiziosa di creare un framework in grado di gestire anche la persistenza dei dati lato server, in modo simile alla soluzione proposta da Knockback (l'evoluzione di Knockout alla quale abbiamo accennato nel paragrafo 2.2.4).

6.1 Limiti attuali e sviluppi futuri

Come prima cosa vogliamo elencare alcuni dei punti che attualmente rappresentano dei limiti di easyHTML ma sui quale è possibile intervenire senza rivoluzionare la struttura di base, così da rendere la libreria uno strumento adatto per creare applicazioni reali.

Per prima cosa sarebbe opportuno gestire nativamente più tipologie di dato non semplici, per esempio i form o le tabelle, che sono utilizzate spesso specialmente nella creazione di applicazioni web. Per questi elementi sarebbe necessario gestire in modo personalizzato anche un maggior numero di eventi (come ad esempio `onSubmit` oppure `onSelectedOption`) in modo da limitare il più possibile gli interventi con jQuery per definire comportamenti non standard una volta aggiunti gli elementi nella View.

Inoltre la gestione di aggiunta e rimozione a runtime presenta qualche lacuna se paragonata allo stato dell'arte (par 5.3.1). Non vogliamo introdurre delle funzioni ad hoc per scorrere array o gruppi di elementi del Model come avviene in Knockout, perchè riteniamo che quello sia un approccio poco intuitivo per il programmatore. Tuttavia c'è la necessità di creare metodi per l'aggiunta e rimozione veloce di Item e ItemAdvanced che fanno parte di gruppi di elementi simili, magari creando una sorta di funzione `H5.runtime.addItem(myArray, myNewItem)`. Oppure si potrebbe estendere la classe Array in modo che, se contiene Item, sia possibile invocare metodi del tipo `myArray.addItem()` e `myArray.removeItem()` con la garanzia di vedere automaticamente aggiornato il grafo delle dipendenze.

Per il resto riteniamo che aggiungere troppe funzionalità non sia proficuo, o per lo meno non nel momento in cui queste sostituiscono soluzioni più immediate e intuitive. Per esempio sarebbe inutile separare float e integer in 2 sObject differenti (era uno degli interventi discussi), poiché la complessità sarebbe maggiore senza ottenere dei guadagni tangibili.

6.1.1 Approccio inverso: creazione manuale della View e generazione automatica del Model

Abbiamo visto che easyHTML obbliga il programmatore a generare anche nel Model tutti gli oggetti da visualizzare nella View, sebbene in alcuni casi essi servano solo per creare collegati con altre fonti nel ViewModel (par 4.2.1). Inoltre gli elementi della View vengono generati automaticamente a partire dalle informazioni che si possono ricavare dagli altri 2 componenti, come il tipo di dato gestito, le modalità d'interazione consentite e lo stato della variabile (par 4.2.3). Unendo queste due circostanze possiamo immaginare come la libreria si presti piuttosto bene a un'interpretazione diametralmente opposta: invece di creare il codice del Model, sistemare il ViewModel e ottenere la View, sarebbe possibile sviluppare un'editor che consenta la creazione della View e parte del ViewModel utilizzando solo un'interfaccia visiva senza scrivere codice (si pensi a come sia possibile comporre gli elementi di un'applicazione con uno strumento come Visual Studio, dove è sufficiente selezionare una tipologia di costrutto da un menu e trascinarlo nell'ambiente di lavoro, definendone poi proprietà e altri dettagli). La creazione di binding sarebbe piuttosto semplice ed efficace in un simile ambiente di lavoro (si pensi a uno strumento che permette di tracciare una freccia da un elemento a un altro per indicare che dipende da esso).

Ovviamente questo porrebbe dei limiti nell'uso di risorse JavaScript; per esempio prendere valori da variabili non presenti nel Model non sarebbe così semplice, poiché potremmo non avere un controllo diretto su di esse (magari perchè sono generate da altre librerie o perchè non abbiamo accesso a tutti i dati dell'applicazione). Tuttavia questo ambiente di sviluppo “from scratch” consentirebbe di generare buona parte del codice per gestire la logica dell'applicazione, col vantaggio di non dover conoscere a fondo la sintassi di easyHTML e di essere utilizzabile anche da programmatori meno esperti. Dopo questa fase si potrebbe dare la possibilità d'intervenire direttamente sul codice prodotto in modo da ottenere anche gli effetti più complicati.

6.1.2 Lato server

La libreria che abbiamo sviluppato semplifica molto l'utilizzo di dati in remoto nel caso in cui sia possibile utilizzare gli WebSocket per comunicare con un server. Tuttavia è necessario che anche quest'ultimo supporti la comunicazione con questo specifico elemento. Talvolta invece non è necessario instaurare un canale di comunicazione permanente, ma basterebbe notificare le modifiche in caso di update. Pensando più in grande potrebbe anche essere impossibile mantenere sul client contemporaneamente tutti gli oggetti del Model, oppure potrebbe essere necessario che essi risiedano permanentemente sul server poiché condivisi da più utenti (che magari possono aggiornarli nello stesso momento). Per evitare problemi di sincronismo sarebbe necessario una gestione molto più approfondita dell'invio e ricezione dei dati in modo che diventi possibile garantire la persistenza degli stessi sul server, e l'eventuale notifica di cambiamenti a tutti gli agenti che stanno interagendo nello stesso momento.

Uno strumento del genere, per essere standardizzato con metodi ad hoc, richiede ovviamente una controparte lato server. L'idea sarebbe quella di creare un framework per permettere di delocalizzare il Model e renderlo condiviso. L'ampliamento delle funzionalità di easyHTML sarebbe delegato in questo caso a un plugin, in modo da non appesantire troppo la libreria con parti che in molti casi sarebbero del tutto superflue (ad esempio un'applicazione che funziona completamente in locale non trarrebbe alcun beneficio da questo approccio, e nemmeno una che richiede dati da un server su cui lo sviluppatore non ha controllo, scenario piuttosto comune).

APPENDICE

Di seguito riportiamo parte della documentazione disponibile online. Evitiamo di riportare la reference per intero poiché sarebbe molto lunga, e ci limitiamo a un estratto delle parte inerenti gli Item e gli sNumber, in modo da presentare i metodi a disposizione del programmatore per alcuni oggetti significativi.

Global objects

Item([struct]): *instantiate new foundalmental Model object*
@param struct: Item structure in JSON format, or another Item to clone (by name or by ref)
@return this new Item.

here(itemName): *decide where Item's content should be displayed in HTML page.*
@param itemName: Item name (or names, or array of names) to display
@return boolean: eventually signals something wrong
@example: here("myItem1", ["myItem3", "myItem4"]);

inside(sonItems, partentItemName)*: attach one or more items inside another item*
@param partentItemName: name of father item where to place other item(s)
@param sonItems: name of son item(s)

@return boolean: eventually signals something wrong
@example: here(myItem);

attachItemInsideId(idIn, itemsNew): *available only at runtime. Attach one or more item inside an already placed HTML element with a specified id*
@param idIn: id of an HTML element
@param itemsNew: item/items names to attach inside HTML element
@return boolean: eventually signals something wrong
@example: attachItemAfterId("HTMLid", "item1toDraw", "item2toDraw");

attachItemAfterId(idIn, itemsNew): *available only at runtime. Attach one or more item after an already placed HTML element with a given id*
@param idIn: id of an HTML element
@param itemsNew: item/items names to attach after HTML element
@return boolean: eventually signals something wrong
@example: attachItemAfterId("HTMLid", "item1toDraw", "item2toDraw");

[...]

Item Methods

Item.addNumber(objIn): *creates a new sNumber object.*
@param objIn: accept one or more strings or objects. string are in the form "name" or "name:value" where value must be a number; object are in the form {"attribute":"value", "aatribute2":"value2", ...} with "name" mandatory
@return Item: Returns this Item.
@example: myItem.addNumber({"name": string, "value": number, "label": string, "modifiable": boolean, "binded": string, "max": number, "min": number, "isInteger": boolean});
@example: myItem.addNumber("gol1", "gol2", "golCasa:4", "golFuori:1", {"name": "totaleGoal", "value": 5});

Item.addString(objIn): *creates a new sString object.*

@param objIn: accept the following combination of objects: string (defines a sString name or its name and its value if in the form "name:value"), object (defines any sString's method; name is mandatory).

@return Item: Returns this item

@example: myItem.addString({name: string, value: string, modifiable: boolean, binded: string});

@example: myItem.addString("squadra1", "squadra2", "inCasa:no", {"name": "risultato", "value": "pareggio"});

Item.addBoolean(objIn): *creates a new sBoolean object.*

@param objIn: accept the following combination of objects: string (defines a sString name or its name and its value if in the form "name:value"), object (defines any sBoolean's method; name is mandatory).

@return Item: Returns this item

@example: myItem.addBoolean({name: string, value: boolean, modifiable: boolean, binded: string});

@example: myItem.addBoolean("squadra1", "squadra2", "inCasa:no", {"name": "risultato", "value": "pareggio"});

Item.addDate(objIn): *creates a new sDate object.*

@param objIn: accept the following combination of objects: string (defines a sDate name or its name and its value if in the form "name:value"), object (defines any sDate's method; name is mandatory).

@return Item: Returns this item

@example: myItem.addDate({name: string, value: "21-06-2012", modifiable: boolean, binded: string});

@example: myItem.addDate("dataNascita", "giornoMatrimonio", {"name": "battesimo", "value": "10-10-1990"});

Item.clone(inputCopy): *make this Item a parameter's Item clone*

@param inputCopy: another Item (different from this) to copy

@return Item: Returns this item

@example: myItem.clone(anotherItem);

Item.setLocalStorage(objIn): *decide if item must be saved to local storage or not*

@param objIn: boolean (true to use localStorage)

@return Item Returns this Item.

@example: myItem.localStorage(true);

Item.setInvisible(objIn): *decide if item should be excluded from visualization*

@param objIn: boolean (true to set invisible)

@return Item: Returns this Item.

@example: myItem.setInvisible(true);

Item.setClass(objIn): *assign class(es) attribute to item's correspondent HTML element (if argument is a string or an array of strings) or to attributes sObject (if passed 2 string as "name_sObject", "attribute" or 1 object with couples "name sObject": "attribute")*

@param objIn: string or array or 2 strings or object (sObject assignment)

@return Item: Returns this Item.

@example: myItem.setClass("importantItem red");

@example: myItem.setClass(["importantItem", "red"]);

@example: myItem.setClass("mySObject": "red");

@example: myItem.setClass({"sObject1": "attr1", "sObject2": "attr2"});

Item.setValue(objIn): *set attribute value of one or more sub-object (they must be sObjects)*

@param objIn: accept 2 strings or 1 object. 2 Strings arecouple "name", "value"; object is made by couples "name":"value"

@return Item: Returns this item.

@example: myItem.setValue({"squadra": "AC Fiorentina", "annoFondazione": 1929});

@example: myItem.setValue("nomeStadio", "Artemio Franchi");

Item.setLabel(objIn): *set attribute label of one or more sub-object (they must be sObject) or set a label to be displayed as item's title*

@param objIn: 1 string (an item label) or 2 strings (couple "name" of sObject and "label" to be set) or 1 object (made by couples "name of sObject to be set": "label")

@return Item: Returns this item.

@example: myItem.setLabel("albergo");

@example: myItem.setLabel({"squadra": string, "stadio": number});

@example: myItem.setLabel("viaStadio", "Stadio in via");

Item.setModifiable(objIn): *set attribute modifiable of one or more sObject*

@param objIn: accept 2 param (1 string + 1 booelan) or 1 object. In first case string is the name, in the second object is made by couple "name": value

@return Item: Returns this item.

@example: myItem.setBoolean({"nuovoGiocatore": true, "italiano": false});

@example: myItem.setBoolean("stadioDiProprieta", false);

Item.hideUpdate(hide): *hide update button of this item*

@param hide: boolean to set hidding on/off (default is on)

@return Item: Returns this item.

@example: myItem.hideUpdate(true);

[...]

sNumber Methods:

sNumber.get(): *get number stored in value property or binded value (eventually execute binding defined function)*

@return number: Returns the stored/binded value or false if used before runtime and binding is with ItemSocket

@example: myItem.myNumber.get();

sNumber.getFather(): *available only at runtime. Return father Item*
@return Item: Returns father item, or false if used before runtime
@example: myItem.myNumber.getFather();

sNumber.setValuefunction(value, [toInt]): *set variable state*
@param value: accpets only number, and they must be lower than acutal max and higher than actual min
@param toInt: boolean value to set float/integer type (true is float, default behaviour)
@return sNumber: Returns this sNumber
@example myItem.myNumber.setValue(25.4);
@example myItem.myNumber.setValue(12, false);

sNumber.setLabel(label): *set a value to be displaied as HTML label in place of variable name*
@param label: a string that indicates variable's content (should be no longer than 3 words)
@return sNumber: Returns this sNumber
@example: myItem.myNumber.setLabel("id Number");

sNumber.setModifiable(modif): *decide if data is modifiable directly from HTML interface.*
@param modif: boolean indicating if value is modifiable (true is modifiable, default behaviour)
@return sNumber: Returns this sNumber
@example: myItem.myNumber.setModifiable(true);

sNumber.setBinding(binding): *set data binding to some destination of value (eventually returned from a custom function)*
@param binding: string defining a binding to another nNumber, ar a function returning a number. Eventual use of other sObject value will guarantee dependancies managing
@return sNumber: Returns this sNumber

@example myItem.myNumber.setBinding("anotherItem.anotherNumber");
@example myItem.myNumber.setBinding(function(({ / any code here */ return some_numeric_value });*

sNumber.attachSocket(binding, [subpart]): *set data binding with ItemSocket and eventually specify a remote source object*
@param binding: string defining a binding to an ItemSocket
@param subpart: name of target object. COuld be used only if a JSON element is expected to be managed by ItemSocket
@return sNumber: Returns this sNumber
@example myItem.myNumber.attachSocket("myItemSocket");
@example myItem.myNumber.attachSocket("myItemSocket", "subOject.myTargetObject");

sNumber.setMin(numMin): *set minimum possible value*
@param numMin: accepts only number, and it must be lower than acutal value and acutal maximum value (if any)
@return sNumber: Returns this sNumber
@example myItem.myNumber.setMin(1);

sNumber.setMax(numMax): *set maximum possible value*
@param numMax: accepts only number, and it must be higher than acutal value and acutal minimum value (if any)
@return sNumber: Returns this sNumber
@example myItem.myNumber.setMax(55);

sNumber.setIsInteger(isInteger): *decide if number must be an integer.*
@param isInteger: boolean indicating if value is integer (false means float, default behaviour)
@return sNumber: Returns this sNumber
@example: myItem.myNumber.setIsInteger(true);

sNumber.setInvisible(invisible): *decide if number must be excluded from visualization*

@param invisible: boolean indicating if must be invisible (false means visible, default behaviour)
@return sNumber: Returns this sNumber
@example: myItem.myNumber.setInvisible(true);

sNumber.setClass(objIn): *set class of corresponding HTML element*
@param objIn: one or more string that represent a class name
@return sNumber: Returns this sNumber
@example myItem.myNumber.setClass("red");

[...]

Bibliografia

1: Samir Bhatnagar, Phil Vincenzes, Importance of User Interface, 2011, http://www.governmentciomagazine.com/2011/12/importance-user-interface

2: Kyle Lutes, Separating the User Interface from the Business Logic, 2001, http://www.informit.com/articles/article.aspx?p=22679

3: Wikipedia, Data consistency, [2012], http://en.wikipedia.org/wiki/Data_consistency

4: Evan Rodgers, Tizen shows off the potential of HTML5 mobile apps, 2012

5: Mauro Notarianni, Mozilla mostra Firefox OS, sistema operativo per smartphone, 2012, http://www.macitynet.it/macity/articolo/Mozilla-mostra-Firefox-OS-sistema-operativo-per-smartphone/aA62768

6: Addy Osmani. Andrée Hansson, Patterns For Large-Scale JavaScript Application Architecture, 2011, http://addyosmani.com/largescalejavascript/

7: Josh Smith, WPF Apps With The Model-View-ViewModel Design Pattern, 2009, http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

8: W3C, HTML5 draft, 2012, http://www.w3.org/TR/html5/

9: Michael Tuck, The Real History of the GUI, 2001, http://www.sitepoint.com/real-history-gui

10: Ghezzi, Jazayeri, Mandrioli, Ingegneria del Software: Fondamenti e Principi, 2004

11: Jonathan Strickland, Is there a Web 1.0?, 2008, http://computer.howstuffworks.com/web-10.htm

12: Masaichiro Yoshioka, Diffusion of Internetand the New Lifestyle, 2004, http://www.wto.org/english/tratop_e/inftec_e/symp_oct04_e/yoshioka_e.pdf

13: David Goldman, The beginning of the end for Adobe's Flash, 2011, http://money.cnn.com/2011/11/10/technology/adobe_flash/index.htm

14: W3C, W3C Targets 2014 for HTML5 Standard, 2011, http://www.w3.org/2011/02/htmlwg-pr.html

15: multipli (wiki), Why is HTML5 better than Flash & Silverlight?, 2009, http://blog.burlock.org/opinion/73-why-is-html5-better-than-flash-a-

silverlight

16: Boris Simus, IMPROVING HTML5 CANVAS PERFORMANCE, 2011, http://www.html5rocks.com/en/tutorials/canvas/performance/

17: Rob Gravelle, Comet Programming: Using Ajax to Simulate Server Push, 2009, http://www.webreference.com/programming/javascript/rg28/index.html

18: Bochicchio, Casati, Civera, Mostarda, Golia, HTML 5 con CSS3 e ECMAScript5, 2011

19: Stephen Chapman, JavaScript Data Typing, 2011, http://javascript.about.com/od/hintsandtips/a/datatype.htm

20: wikipedia, Javascript features, [2012], http://en.wikipedia.org/wiki/JavaScript#Features

21: John Resig, ECMAScript 5 Strict Mode, JSON, and More, 2009, http://ejohn.org/blog/ecmascript-5-strict-mode-json-and-more/

22: John Resig, The World of ECMAScripthttp://ejohn.org/blog/the-world-of-ecmascript/, 2007, http://ejohn.org/blog/the-world-of-ecmascript/

23: Web Tecnology Surveys, Usage of JavaScript libraries for websites, 2012

24: Janus Boye, Why use CSS?, 1998, http://www.irt.org/articles/js135/

25: W3C, CSS3 Reccomandation, 2012, http://www.w3.org/Style/CSS/current-work

26: Greg Sterling, It's A "Post PC" World: Smartphones, Tablets Outpace Traditional PC Growth, 2012, http://marketingland.com/its-a-post-pc-world-smartphones-tablets-outpace-traditional-pc-growth-8951

27: Adrien de Pierres, Cross-Platform Development Considerations, 2011, http://mobile.tutsplus.com/articles/theory/cross-platform-development-considerations/

28: Chris Horton, Is HTML5 the Future of App Development?Read more at http://www.business2community.com/mobile-apps/is-html5-the-future-of-app-development-0138766#h4sUO0x1FGTBVODF.99, 2012, http://www.business2community.com/mobile-apps/is-html5-the-future-of-app-development-0138766

29: Ronald Widha, HTML5: The Future of Web Development, 2012

30: Patrick Hynds , Windows 8 and HTML5: A story just unfolding, 2012, http://www.sdtimes.com/link/36627

31: David Flanagan, JavaScript: The Definitive Guide, 2011

32: Wikipedia, Software design pattern, [2012], http://en.wikipedia.org/wiki/Software_design_pattern

33: Amir Salihefendic, Model View Controller: History, theory and usage, 2011,

http://amix.dk/blog/post/19615

34: Luke Siedle, Backbone Interfaces, 2012, http://lukesiedle.me/using-backbone-interfaces-for-backbone-js/

35: Steven Sanderson, Rich Javascript MVC user interfaces with jMVC, 2007, http://blog.stevensanderson.com/2007/10/04/rich-javascript-mvc-user-interfaces-with-jmvc/

36: Martin Fowler, The Presentation Model Design Pattern, 2006, http://martinfowler.com/eaaDev/PresentationModel.html

37: Steve Anderson, KnockoutJS Presentation, 2011, http://channel9.msdn.com/Events/MIX/MIX11/FRM08

38: W3Techs, Usage statistics of Knockout for websites, 2012

39: Open source, Knockback.js, 2012, http://kmalakoff.github.com/knockback/

40: Bjorn Enki, Separating presentation from content, 2009, http://www.bjornenki.com/blog/separating-presentation-from-content-css

41: George Ornbo, DOM + CSS = a Beautiful Couple, 2006, http://shapeshed.com/dom_css_a_beautiful_couple/

42: W3School, Web Statistics and Trends, 2012

43: Wikipedia, MVVM: Pattern description, [2012], http://en.wikipedia.org/wiki/Model_View_ViewModel#Pattern_description

44: Addy Osmani, Understanding MVVM – A Guide For JavaScript Developers, 2012, http://addyosmani.com/blog/understanding-mvvm-a-guide-for-javascript-developers/

45: Stoyan Stefanov, 3 ways to define a JavaScript class, 2006, http://www.phpied.com/3-ways-to-define-a-javascript-class/

46: Christian Heilmann, Local Storage And How To Use It On Websites, 2010, http://coding.smashingmagazine.com/2010/10/11/local-storage-and-how-to-use-it/

47: Jeremy Likness, Model-View-ViewModel Explained, 2010, http://www.wintellect.com/CS/blogs/jlikness/archive/2010/04/14/model-view-viewmodel-mvvm-explained.aspx

48: Wikipedia, Asynchronous JavaScript and XML, [2012], http://it.wikipedia.org/wiki/Asynchronous_JavaScript_and_XML

49: Robert Nyman, Introducing HTML5 Web Sockets – Taking Bidirectional Communication On The Web, 2010, http://robertnyman.com/2010/10/22/introducing-html5-web-sockets-taking-bidirectional-communication-on-the-web-to-the-next-level-2/

50: JSPerf, JavaScript performance playground, [2012], http://jsperf.com/

51: Devon Govett, jsPerf: The Github of JavaScript Performance Testing,

2010, http://badassjs.com/post/893443459/jsperf-the-github-of-javascript-performance-testing

52: Jan Fajfr, KnockoutJS and Google Maps binding, 2012, http://www.codeproject.com/Articles/351298/KnockoutJS-and-Google-Maps-binding