Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica per il Management

PORTING DI UNA APPLICAZIONE WEB DA MONOLITICA A COMPONENT-BASED

Relatore:
Chiar.mo Prof.
FABIO VITALI

Presentata da:
ANDREA RAGO

I Sessione
A.A. 2020-21

Indice

Introduzione

La continua evoluzione delle tecnologie e dei linguaggi utilizzati per la realizzazione di applicazioni web e la scoperta di nuove vulnerabilità, fa sì che sia necessario svolgere continue attività di manutenzione, aggiornamento e adattamento del preesistente sistema alle nuove tecnologie pubblicate. La possibilità di suddividere un software in aree di diversa competenza e consentirne l'interoperabilità tra loro grazie a layer aggiuntivi di software (che nel seguente elaborato verranno chiamati "intermediari"), fa sì che l'accoppiamento tra le parti sia più debole e che i singoli moduli possano essere aggiornati, modificati o addirittura sostituiti facilmente, senza doversi preoccupare delle dipendenze software di ciascun lato. È importante stabilire solo l'insieme delle informazioni necessarie alla comunicazione tra i vari moduli, al fine di fornire a ciascun lato le informazioni di cui ha bisogno per raggiungere un determinato obiettivo. Vista la crescente complessità delle applicazioni, la sempre maggiore richiesta di scalabilità e tempesitività di risoluzione degli errori, fa sì che la classica architettura monolitica risulti non essere più adatta, favorendo architetture software (come quella component based) che suddividano il software in componenti più piccole, in grado di raggruppare problemi simili e rendendo così la gestione del codice più facile.

Il caso LIME

LIME è un web editor Open Source, sviluppato dal CIRSFID e Università di Bologna e caratterizzato da una vasta possibilità di personalizzazione. LIME guida l'utente alla trasformazione di documenti non strutturati

in documenti ben formati e strutturati in un linguaggio XML based scelto dall'utente (caso più comune sono i documenti di legge). Si compone di due parti; una parte client-side, eseguita esclusivamente sul browser dell'utente e una parte server-side, eseguita su un server remoto. Senza entrare nel dettaglio delle tecnologie usate per la realizzazione di LIME (se ne parlerà adeguatamente nell'apposito capitolo), basti dire che:

- La parte client-side (logica dell'applicazione e view) è realizzata usando il framework ExtJS e la libreria TinyMCE. Nell'arco del tempo le due librerie sono state aggiornate ed ExtJS, non essendo più disponibile liberamente, risulta non essere più compatibile con la natura Open Source di LIME. Inoltre, il forte accoppiamento tra servizi di TinyMCE e i componenti di ExtJS ha fatto sì che nel corso del tempo la manutenzione e aggiornamento del codice divenisse sempre più complessa. Inoltre, non è possibile sostituire la view con altri framework.

- La parte server-side è realizzata utilizzando Node.js come server-proxy per le richieste e PHP per la realizzazione di altri servizi come convertitori e parsers.

La sostituzione di ExtJS, che si occupava di realizzare anche la view, ha reso necessario progettare una nuova architettura per LIME tale da renderla modulare, scorporandone le varie parti e facilitarne la manutenzione e aggiornamenti futuri. Inoltre, la riprogettazione di LIME consentirà al team di sviluppo di sostituire facilmente il framework che realizza la view senza dover realizzare nuovamente i componenti che accedono ai servizi. Prima di continuare, è doveroso fare una precisazione. L'analisi svolta in questo elaborato si concentrerà sulla parte client-side. In particolare, sulla realizzazione di una componente che consente di rendere trasparente le logiche dei servizi alla parte view dell'applicazione.

Capitolo 1

Da architettura monolitica ad una component based

In questo primo capitolo si illustra lo stato dell'arte di LIME, si discutono pregi e difetti di un'architettura monolitica (client-side) che lo caratterizza, descrivendo quali problematiche si possono riscontrare nel tempo e che favoriscono l'adozione di una differente architettura. L'architettura monolitica ha rappresentato per molti anni l'architettura più tilizzata per la costruzione di applicazioni ed è caratterizzata dal fatto che le componenti del software costituiscono un unico blocco di codice, non eseguibile separatamente e condividono le risorse della macchina nel quale avviene l'esecuzione. L'architettura monolitica, nel tempo, è diventata inadatta a fronteggiare le moderne necessità di sviluppo. Infatti [PMA16]:

- In molti casi l'intera applicazione è costruita grazie all'utilizzo di un unico framework. Si crea tra i due un legame indissolubile che rende il framework insostituibile;
- La comparsa di un errore di esecuzione in una porzione dell'applicazione potrebbe portare l'intero sistema a non essere più fruibile;
- Con l'aumentare del numero di funzionalità aumenta anche la complessità del codice e aumentando la complessità del codice aumenta

la difficoltà di comprensione dello stesso, inficiandone la possibilità di manutenzione;

- Si allungameno i tempi di deployment, a causa al fatto che l'intera applicazione deve essere ricompilata e pubblicata anche in seguito ad un piccola modifica;
- Il codice di un'applicazione monolitica è fortemente accopiato e questo ne inficia la manutenzione, scalabilità e deployment.

Un altro problema caratteristico di questa architettura è legato più ad un fattore umano che ad un fattore tecnico. Lo sviluppo di un'applicazione monolitica non comporta particolari scambi di informazioni tra coloro che si occupano dello sviluppo. In tal caso lo sviluppatore potrebbe essere portato a produrre una quantità insufficiente di documentazione o a non produrne affatto, come dimostrato nel caso descritto in "From Monolithic to microservices, An Experience Report from the banking domain"[BDD18] l'architettura monolitica ha fatto sì che gli sviluppatori del precedentente sistema non producessero sufficiente documentazione, tanto da costringere i nuovi sviluppatori a dover ispezionare il codice o a rivolgersi direttamente a coloro che lo hanno sviluppato. L'architettura component-based è un'architettura software che ha l'obiettivo di scomporre lo sviluppo del software in componenti di complessità minore i quali comunicano tra loro grazie a delle interfacce ben definite, che stabiliscono parametri, eventi e metodi al fine di presentare allo sviluppatore il più alto livello possibile di astrazione. Ciascun componente consente allo sviluppatore di riutilizzare codice già scritto per risolvere lo stesso problema in più parti dell'applicazione, semplificandone notevolmente l'aggiornamento e la manutenzione. Inoltre, consente di soddisfare le necessità sempre crescenti di scalabilità e rapido sviluppo di nuove funzioni [PMA16]. Ogni componente deve essere [2]:

- Estensibile per differenziarne le funzionalità;
- Sostituibile con altri componenti che svolgono le stesse mansioni, senza che il resto dell'applicazione ne risenta;

- Incapsulabile in modo da nascondere tutti i dettagli implementantivi del componenente all'utilizzatore esterno;
- Indipendente, cioè dipendere meno possibile da altri componenenti;
- Riutilizzabile in modo che lo stesso problema possa essere risolto in più parti richiamando semplicemente il componente;
- Non dipendente dal contesto, cioè in grado di assolvere il compito senza conoscere il contesto della richiesta che verrà fornito al componente quando chiamato.

La conversione di un'applicazione da monolitica a componenent based e quindi il processo decisivo che stabilisce quali funzionalità raggruppare e come raggrupparle, si basa principalmente sulla seguente domanda: "se due funzioni svolgono compiti simili, possono essere unite in un unico componente?" [DEL19] La possibilità di suddividere il codice in componenti più piccoli fa sì che un gruppo di sviluppatori si possa concentrare esclusivmanente sullo sviluppo e test del singolo componente [CLC05]. Non solo, i vantaggi di un'architettura component based sono molteplici:

- Facilità di distribuzione: la distribuzione di una versione aggiornata di un componente non coinvolge l'intera applicazione;
- Facilità di sostituzione: la possibilità di sostituire un componente a patto di non modificarne l'interfaccia, riduce i costi, semplifica lo sviluppo e l'aggiornamento;
- Aumento dell'affidabilità dell'intero sistema: grazie alla possibilità di riutilizzare i componenti in più parti, fa sì che l'affidabilità del sistema aumenti di pari passo con l'affidabilità del componente.

1.1 Framework ExtJS

Come indicato in introduzione, LIME client-side è realizzato utilizzando il framework ExtsJS e l'editor HTML TinyMCE. ExtJS è un framework

realizzato con pattern architetturale Model View Controller che consente di sviluppare applicazioni web complesse. Oltre a fornire utili strumenti per realizzare la logica di un'applicazione, fornisce anche molti widget (quali, ad esempio, calendari, tabella o pulsanti) facilmente utilizzabili dallo sviluppatore per la realizzazione della view dell'applicazione. Dalla versione 4.1 ExtJS non risulta più accessibile liberamente e pertanto si è reso necessario sostituirlo con un framework differente.

1.2 Editor TinyMCE

TinyMCE è un editor HTML WYSIWYG web based, realizzato con JavaScript, compatibile con la maggior parte dei browser e in grado di convertire una qualsiasi area di testo HTML in un editor di testo con funzionalità avanzate. LIME integra TinyMCE per realizzare le funzioni di editing dei documenti e lo integra in maniera intelligente. Infatti, TinyMCE viene incapsulato all'interno di un componenente realizzato per ExtJS, che ne conserva tutte le funzionalità e consente di sostituire TinyMCE con un altro editor WYSIWYG Open Source. Questo approccio verrà ripreso nella nuova architettura.

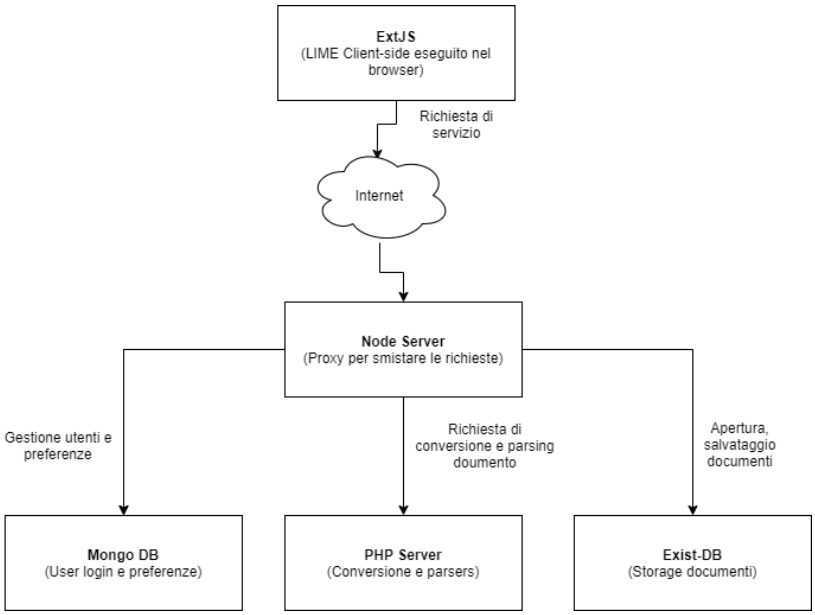
1.3 Node.jS

Node.js è un software server-side, Open-Source, basato sul motore JavaScript V8 di Google ed in grado di eseguire script JS lato server. Node.js ha una gestione delle richieste particolamente efficiente. Anzichè basare l'esecuzione delle richieste su un modello a processi concorrenti, basa l'esecuzione in base alla ricezione di notifiche dal sistema operativo (architettura I/O event-driven). Alla ricezione di una notifica, vengono eseguite le operazioni preposte alla richieste e una volta che il risultato è pronto, viene eseguida la funzione di callback corrispondente. La particolare efficienza di Node.js lo rende ideale in contesti di applicazioni ad inteso traffico di rete (grazie alla

sua ottimizzazione del Throughput) e in contesti di comunicazioni in tempo reale. A differenza di altri linguaggi di programmazione interpretati (come PHP), Node.js è in grado di gestire la richiesta successiva prima ancora che la precedente sia terminata, eliminando così i tempi di attesa. Al contario, ad esempio PHP, prima di poter essere disponibile a gestire la richieste successiva, deve prima eseguire la precedentente, presentare il contenuto al client e infine eseguire la successiva.

1.4 L'architettura di LIME client-side

LIME è un'applicazione web, client-side eseguita all'interno del browser dell'utente ed ospitata su un server web. La logica dell'applicazione è contenuta esclusivamente client-side e i servizi di storage management, user management, parsing e conversione sono realizzati mediante servizi in esecuzione su server remoti e raggiungibili tramite un server realizzato con tecnologia Node.js (nella fattispecie, la parte server side di LIME). Ogni istanza di LIME può essere personalizzabile configurandola in maniera differente, semplicemente indicando tutte i settaggi all'interno di un file JSON. L'attuale architettura di LIME si articola su tre livelli, come mostrato in figura qui sotto:

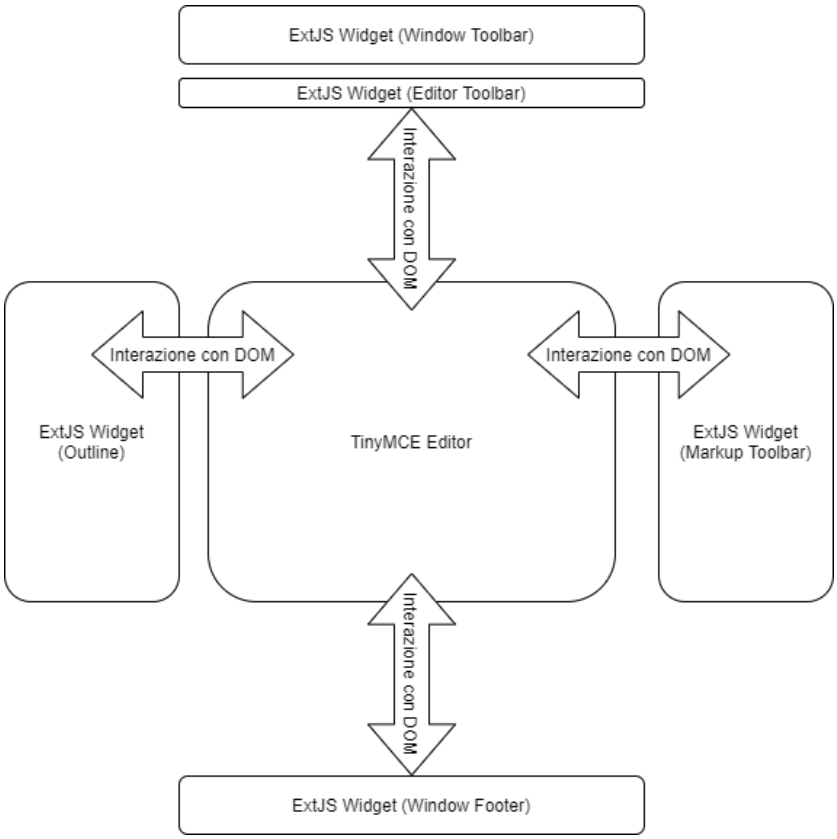


Livelli dell'architettura di LIME

Ad ogni richiesta di servizio - come ad esempio conversione di un documento, salvataggio o login - il client di LIME invia una richiesta al remoto Node.js che in base alla richiesta ricevuta inoltra il compito al server più idoneo all'esecuzione. Il tutto avviene tramite comunicazioni via protocollo HTTP. Come già anticipato nell'introduzione di questo elaborato, l'attuale versione di LIME presenta una criticità non trascurabile che nel tempo ne ha inficiato la capacità di aggiornamento e manutenzione.

1.5 La fase di inizializzazione e l'editor come parte centrale di LIME

LIME è composto da un set di componenti ExtJS e l'editor svolge un ruolo centrale. In fase di inizializzazione, l'editor viene creato, vengono creati i menu e la toolbar principale. Lo schema che segue mostra come ad ogni modifica al documento, l'editor lancia degli eventi che vengono catturati e gestiti dei componenenti dell'applicazione.



Questo approccio fa sì che i componenti di ExtJS accedano direttamente ai servizi di TinyMCE intersecando codice di ExtJS con codice di TinyMCE. Inoltre, ExtJS costituisce fulcro essenziale per l'applicazione LIME in quanto realizza tutto (dalla vista, alla logica dei servizi client-side) rendendo ExtJS insostituibile, se non riscrivendo tutta l'applicazione client-side.

Capitolo 2

Integrazione di intermediari in LIME

Laddove si desidera progettare o riprogettare un'applicazione web al fine di garantirne la massima modularità possibile si rende necessario ideare una nuova architettura composta da gruppi che racchiudano componenti di simile natura. Ogni gruppo dialoga con le altre parti tramite entità che nel seguente elaborato vengono chiamate *intermediari*, i quali consentono di disaccoppiare il dialogo tra gruppi. In tal senso, la modifica o la sostituzione di uno dei due gruppi non costituisce problema per l'altro lato, a patto che i dati scambiati non cambino struttura e natura e venga manutenuta l'interfaccia di comunicazione. Il numero di canali di comunicazione tra gruppi di elementi può variare da 1 a n, in base alle esigenze di sviluppo e in base alla complessità del progetto.

2.1 Definizione di intermediario

In questa sezione si desidera chiarire che cos'è un intermediario, in un concetto più generale, utile a comprendere la nuova architettura del software preso in esame. Per intermediario si intende un'entità che si frappone tra

due soggetti e che ha funzione di passaggio, unione e collegamento tra i due.

Questo modello permette alle entità di rivolgersi ad altre entità del sistema in modo trasparente, conoscendo soltanto l'insieme delle informazioni richieste dall'intermediario e specificando il destinatario di tale richiesta. Sarà poi premura di quest'ultimo di controllare e adattare le informazioni per l'entità di destinazione e inoltrare il messaggio al destinatario specificato. Viceversa, l'intermediario si occupa di adattare e controllare le informazioni prodotte e inoltrare il risultato all'entità iniziale. L'intermediario è l'unica delle entità coinvolte nello scambio a dover conoscere tutte le informazioni necessarie ad entrambi i lati affinchè lo scambio delle informazioni possa svolgersi correttamente. Questo fa sì che il legame tra le due entità (quella mittente e quella di destinazione) sia più debole e facilmente adattabile a future modifiche. Inoltre, nel caso in cui il destinatario specificato non fosse disponibile, l'intermediario può decidere di inoltrare la richiesta ad un destinatario sostituto e di riconsegnare il risultato prodotto al mittente, senza che quest'ultimo sia a conoscenza dello scambio. Di seguito un esempio per chiarire ulteriormente il concetto. Si supponga che metodo doSomething() della classe GenericService:

```
class GenericService {
    // .. altra porzione di classe ...
    static doSomething( ...params){
        // funzione effettua delle operazioni
        // e restituisce il risultato
    }
    // .. altra porzione di classe ...
}
```

Venga richiamato dalle tre funzioni:

```
function entita1(){
    var result = GenericService.doSomething( ...params );
}
function entita2(){
    var result = GenericService.doSomething( ...params );
}
function entita3(){
    var result = GenericService.doSomething( ...params );
}
```

Si supponga ora che lo sviluppatore decida che il metodo doSomething() debba essere deprecato e sostituito da un nuovo metodo più robusto.

```
class GenericService {
    // .. altra porzione di classe ...
    static doSomething( ...params){
        // il metodo non fa piu' nulla e avvisa lo sviluppatore
        // lanciandoo un'eccezione
    }
    static newDoSomething( ...params){
        // funzione effettua delle operazioni
        // e restituisce il risultato
    }
    // .. altra porzione di classe ...
}
```

In assenza di un intermediario, sarebbe necessario intervenire in ogni parte del codice dove viene chiamato il vecchio metodo e sostituirlo con il nuovo. L'introduzione di un intermediario, disaccoppia le tre funzioni dalla classe GenericService, permettendo allo svilupattore di sostituire il vecchio metodo doSomething() soltanto all'interno del metodo intermediario:

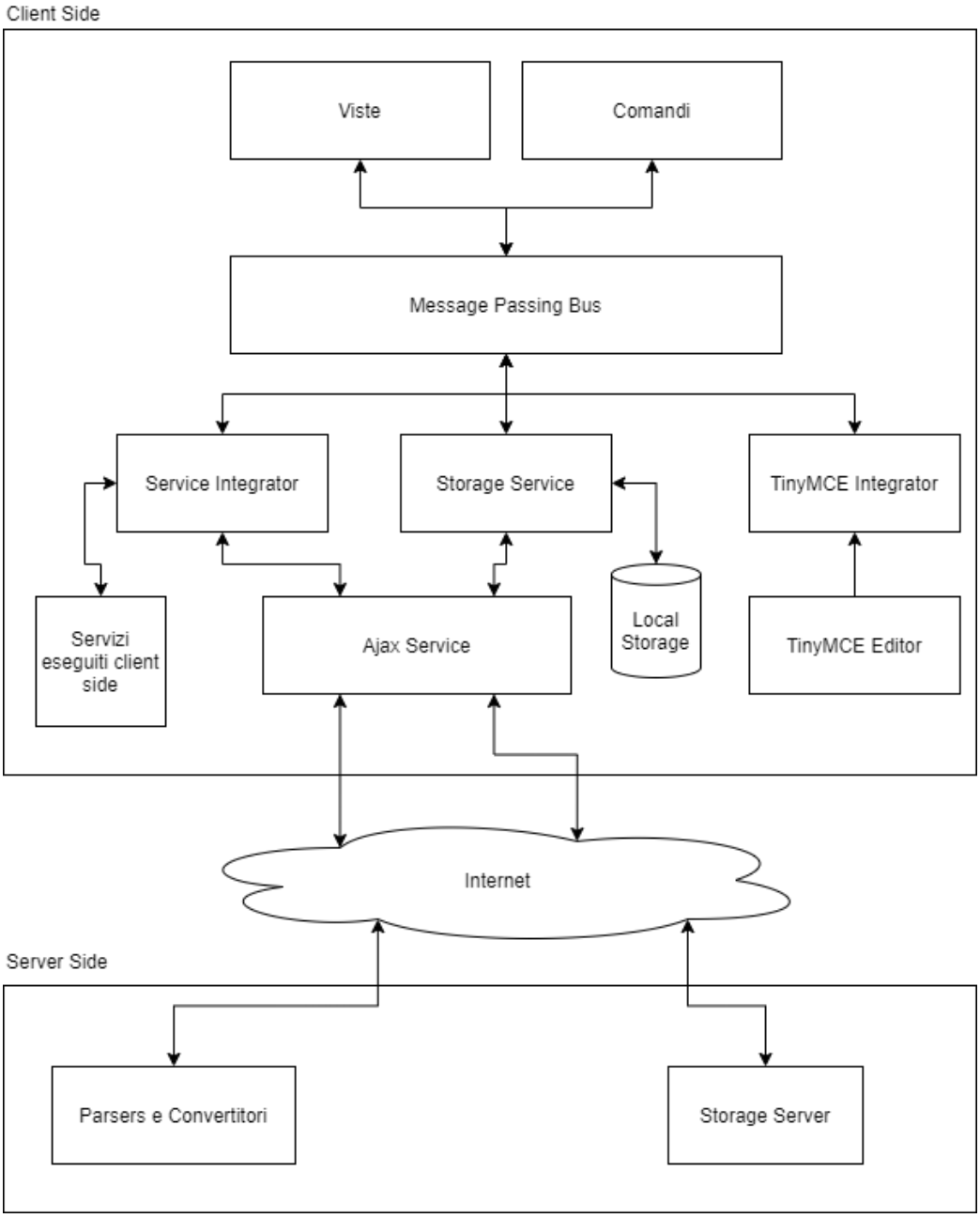
```
function intermediary(..params){
    // chiamo una solta volta il nuovo metodo
    return GenericService.newDoSomething(...params);
}
// le tre funzioni di prima richiama la funzione intermediary
function entita1(){
    var result = intermediary( ...params );
}
function entita2(){
    var result = intermediary( ...params );
}
function entita3(){
    var result = intermediary( ...params );
}
```

2.2 La nuova architettura per LIME

I nuovi requisiti e di conseguenza la riprogettazione dell'architettura ha affrontato due problemi:

- Il primo relativo al forte accoppiamento tra i servizi di ExtJS e TinyMCE e con obiettivo di rendere TinyMCE sostituibile;
- Il secondo relativo all'intercambiabilità della view in modo tale che, in futuro, la view possa essere sostitutita agilmente senza dover riscrivere la logica dei servizi in base al nuovo framework.

Infine, sono stati separati i componenenti della vista e i componenti che non richiedono la visualizzazione a schermo di alcun elemento (come i servizi), così da risolvere il problema di forte accoppiamento tra i servizi di ExtJS e i servizi di TinyMCE.



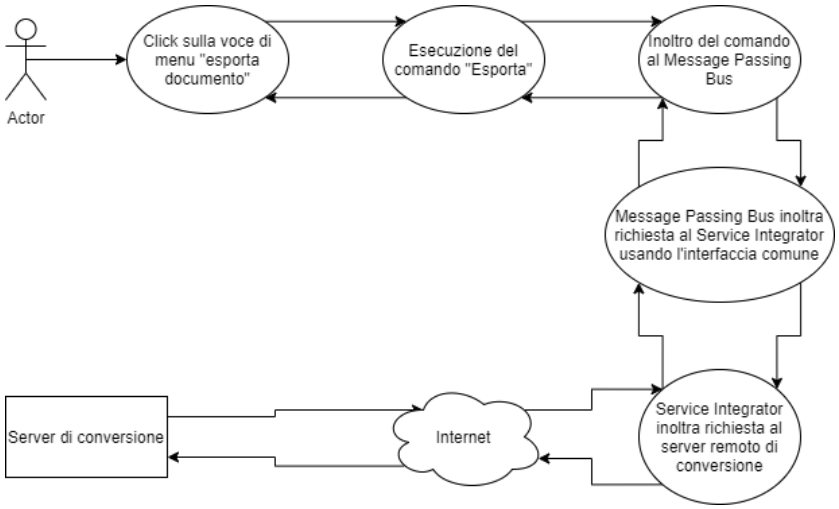
Prima di proseguire si illustra brevemente l'architettura mostrata nella figura sopra al fine di fornire maggiori informazioni per comprendere i passaggi successivi:

- Viste: costitusicono l'insieme degli elementi che occupano una parte

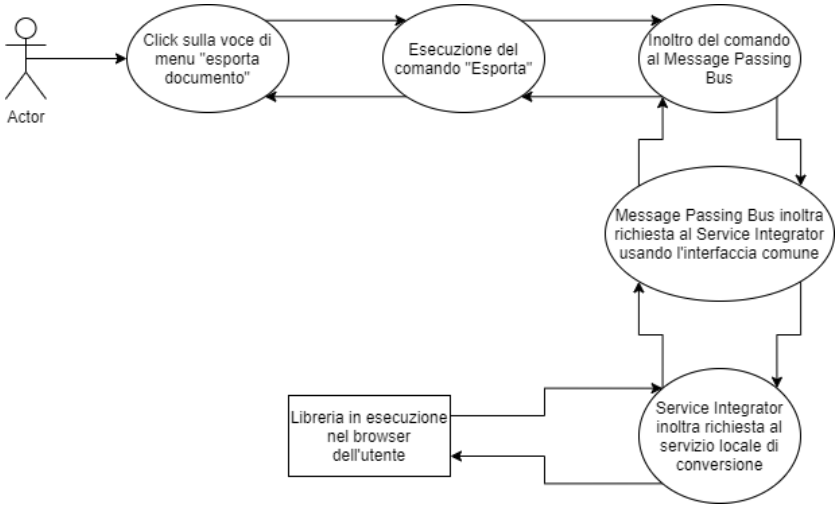
visibile all'utente ed espongono le proprie funzionalità tramite comandi ed eventi;

- Comandi: costitusicono l'insieme degli elementi che consentono all'utente di svolgere una determinata azione, siano essi un pulsante o un menu (es: menu di apertura file salvato, per aprire una vist o formattare il testo). A ciascun comando corrisponde il lancio di un evento e a tale lancio corrisponde l'esecuzione di due callback (una before callback e una after callback, eseguite rispettivamente prima e dopo il lancio del comando);
- Message Passing Bus: canale di comunicazione principale di tutti i componenti;
- ServiceIntegrator, Storage Service e TinyMCE Integrator: fungono da intermediari e costituiscono le entità alla quali il Message Passing Bus inoltra le richieste provenienti dalla view;

L'architettura si organizza intorno al Message Passing Bus che costuisce l'unico canale di comunicazione tra le varie parti del sistema. Ciascun comando della web application dialoga con esso in modo che non sia interpellato il diretto destinatario di tale richiesta, ma rivolge la richiesta al Message Passing Bus che inoltra la richiesta al destinatario opportuno (ServiceIntegrator o StorageService) che a sua volta svolge ruolo di intermediario verso il destinatario più ideneo a soddisfare la richiesta. Ad esempio, considerando l'intento dell'utente di esportare il documento che sta visualizzando a schermo in formato PDF. All'intenzione dell'utente corrisponde un click su una voce di menu, la quale richiama il rispetttivo comando. A questo punto, il comando di esportazione file non rivolge la richiesta direttamente al servizio di esportazione file, ma la richiesta viene consegnata al ServiceIntegrator che la inoltra al destinatario opportuno, sia esso un server remoto oppure un servizio eseguito client-side. In questo modo il comando eseguito riceve lo stesso risultato in maniera completamente trasparente, senza conoscere la posizione del destinatario della richiesta.



Richiesta conversione PDF eseguita su server remoto



Richiesta conversione PDF eseguita localmente

2.3 Intercambiabilità della vista

La realizzazione dell'intermediario si pone anche l'obiettivo di rendere intercambiabile il framework che realizza la view dell'applicazione in mondo da renderlo sostituibile con altri framework, concentrando lo sviluppo esclusivamente sulla realizzazione della view, senza doversi occupare dei componenti che realizzano i servizi necessari al funzionamento dell'applicazione. Per raggiungere questo livello di modularità si è deciso di introdurre gli intermediari:

- ServiceIntegrator: fornisce un'interfaccia comune per richiedere l'esecuzione di servizi (es: servizi di conversione, parsing, ecc) e mascherarne la logica, in modo che possano essere spostati server-side o client-side - in base a criteri di efficienza, facilità di sviluppo e implementazione - in maniera trasparente per la view.

- StorageService: progettato con logica simile al ServiceIntegrator. Fornisce un'unica interfaccia comune per richiedere il salvataggio, l'apertura e la navigazione dello storage che contiene i file scritti dall'utente. Con l'obiettivo di preservare il contenuto del documento qualora non fosse possibile salvarlo nel file system designato, il StorageService in futuro prevederà il salvataggio del documento nel Local Storage[1]del browser in maniera trasaparente al resto dell'applicazione.

- AjaxService: costituisce un'interfaccia comune a ServiceIntegrator e StorageService per consentire ai servizi di interagire con risorse remote. Gestisce in maniera trasparente agli utilizzatori aspetti di autenticazione, gestione della sessione e degli errori.

[1]Local Storage: spazio della sessione di un browser nel quale i dati memorizzati non hanno scadenza

Capitolo 3

Implementazione degli intermediari in LIME

In questo capitolo si parlerà più nel dettaglio delle classi ServiceIntegrator, StorageService e AjaxService oggetti principali dello studio di questo elaborato e realizzati sul modello di un event handler (ad esclusione di AjaxService). Insieme costituiscono la libreria di richiesta servizi per LIME e nel contempo una libreria utilizzabile in qualsiasi framework utilizzi TypeScript o in grado di includere script JavaScript. ServiceIntegrator è ideale per la registrazione e lancio di eventi, mentre StorageService è piú specifico per registrare eventi di gestione di file e/o filesystem.

3.1 L'oggetto Promise in Javascript e TypeScript

JavaScript è un linguaggio di programmazione orientato agli oggetti, interpretato (quindi non compilato prima dell'esecuzione) e basato sugli eventi. Dapprima utilizzato esclusivamente per la programmazione client-side (successivamente per la programmazione server-side), JavaScript consente di rendere le pagine web HTML dinamiche. L'esecuzione delle funzioni ideate dallo sviluppatore avviene soltanto in seguito al verificarsi di un evento sca-

tenato dall'utente che interagisce con la pagina web. Tali eventi spaziano dal termine del caricamento della pagina, al click dell'utente su un pulsante o su un link. JavaScript venne introdotto durante gli anni '90 da Sun e Netscape e divenne famoso in poco tempo perchè consentiva di manipolare il contenuto della pagina web senza richiedere esecuzione di alcuna istruzione lato server. Nel 1997 JavaScript venne standardizzato da EcmaScript International e negli anni successivi ha subito modifiche ed estensioni. Nel 2015 viene standardizzata la versione ES6 (EcmaScript 2015) e successivamente, nel 2019, la versione 10 (EcmaScript 2019). Sin dagli albori JavaScript fu progettato per essere eseguito nello stesso processo nel quale viene eseguito il browser. Tale caratteristica non permette a JavaScript di fornire un ambiente ideale per l'esecuzione di operazioni lunghe e potenzialmente bloccanti, pena il congelamento del browser. Con il passare degli anni la complessità dei siti, non più solo utilizzati per leggere informazioni, ma sempre come più complesse applicazioni, portò ad una maggiore necessità di interattività delle pagine web. Così nei primi anni 2000 l'introduzione del modello AJAX (Asynchronous JavaScript and XML) facilitò la realizzazione applicazioni web client-side e server-side fortemente interattive, in grado di minimizzare il traffico di rete e di modificare il DOM senza dover richiedere nuovamente l'intera pagina al server e di supportare, proprio grazie alle operazioni asincrone, l'esecuzione di azioni più lunghe e che potenzialmente potevano bloccare il browser dell'utente. La gestione dei tipi di dato in JavaScript è diversa dai più tradizionali linguaggi di programmazione orientati agli oggetti. Infatti, JavaScript non prevede controlli statici sui tipi di dato, ma effettua una conversione implicita tra essi. Questo significa che non sono le variabili ad essere tipate, ma lo sono sono i dati stessi ed eventuali incompatibiltà vengono riscontrate soltanto a runtime. Questa libertà concessa da JavaScript, in casi di applicazioni complesse, potrebbe risultare un ostacolo nella risoluzione di problemi a runtime. Nel 2012 Microsoft introduce TypeScript, un linguaggio di programmazione che estende JavaScript e introduce il controllo statico sui tipi di dato. Gli script realizzati in TypeScript non possono essere eseguiti direttamente nel

browser, ma devono essere compilati (da un transpiler) in codice JavaScript eseguibile direttamente dal browser. Non a caso si parla di Typescript. Infatti, il ServiceIntegrator, StorageService, AjaxService e tutti i servizi connessi sono stati realizzati in TypeScript, beneficiando così di tutti gli aspetti positivi derivati dall'utilizzo di un compilatore, tra i quali controllo sui tipi di dato durante la compilazione e non a run-time e implementazione di interfacce, per definire uno scheletro di un modello da implementare e definire successivamente in una classe. L'oggetto Promise fornisce un utile strumento per ottenere dati in modo asincrono e non immediatamente disponibile al momento della loro creazione [5]. Assumono differenti stati in base all'esito dell'operazione che devono svolgere:

- Pending: stato inziale, la Promise non nè soddisfatta, nè respinta;
- Fullfilled: stato finale raggiunto se e solo se la Promise è stata soddisfatta e i dati richiesti sono stati ottenuti senza errori;
- Rejected: stato finale raggiunto se e solo se la Promise è stata respinta in seguito al verificarsi di un errore.

Una volta raggiunto lo stato Fullfilled o Rejected, la Promise non può che ritornare nello stato Pending. La classe Promise preve un costruttore con un unico parametro corrispondente ad una funzione (handler) che ha il compito di gestire lo stato di successo o insuccesso della Promise.

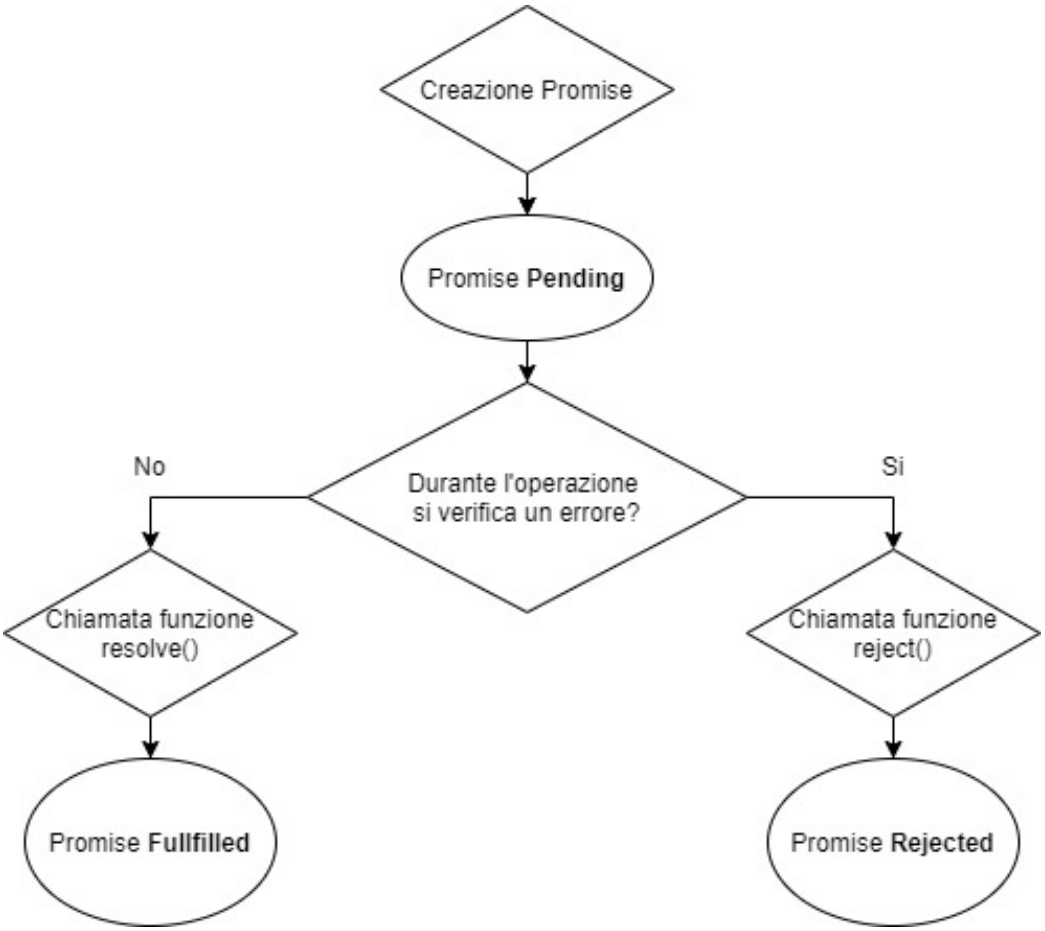


Diagramma degli stati di una Promise

Nell'esempio che segue, le funzioni resolve() e reject() sono rispettivamente invocate se la Promise viene soddisfatta oppure respinta.

```
var promise = new Promise(function(resolve, reject) {
    try {
        // svolge operazione, nessun errore
        // Promise soddisfatta, chiama funzione resolve
        resolve(risulato);
    } catch (errore){
        // errore riscontrato durante l'operazione
        // Promise rigettata, chiama funzione reject per gestire errore
        reject(errore);
    }
});
```

Oppure:

```
var promise = new Promise(function(resolve, reject) {
    if( condizione ) {
        // se la condizione verificata e' vera
        // Promise soddisfatta, chiama funzione resolve
        resolve(risulato);
    } else {
        // errore riscontrato durante l'operazione
        // Promise rigettata, chiama funzione reject per gestire errore
        reject(errore);
    }
});
```

Alle funzioni resolve() e reject() corrispondono due funzioni che lo sviluppatore può creare per ottenere il valore della Promise (nel caso di Promise fullfilled) oppure per gestire l'errore (in caso di Promise rejected). Questo approccio maschera completamente l'esigenza di capire se l'evento richiamato

dal ServiceIntegrator o StorageService è sincrono o asincrono. Lo sviluppatore deve semplicemente indicare due callback da eseguire per gestire i due casi. Questo è possibile grazie alla catena di then():

```
var promise = new Promise( // parametri della promise );
promise.then(
    function(result){
        // operazioni eseguite con il valore
        // ottenuto dalla promise fullfilled
        // result contiene risultato operazione
    },
    function(error){
        // operazioni da eseguire per gestire errore
        // se promise rejected
        // error contiene valore errore
    }
);
// per intercettare qualunque errore
promise.catch({
    // gestione degli errori
});
```

La scelta di parlare delle Promise non è casuale. Infatti, essendo gli eventi richiamati dal ServiceIntegrator e dal StorageService per la maggior parte asincroni, le Promise hanno rappresentato un utile strumento per realizzare le funzioni previste fornendo al Message Passing Bus due utili agganci (le callback) per la gestione degli errori da presentare alla view (l'implementazione è visibile nell'apposito capitolo).

3.2 La API *Fetch*

La API *Fetch* fornisce un'interfaccia di accesso per scaricare risorse, locali o remote tramite rete. Realizza una comunicazione di tipo client-server alla cui richiesta corrisponde una risposta. La Fetch API si realizza in un metodo fetch(url, init) che prevede due parametri:

- Il primo parametro obbligatorio (url) corrisponde all'URL della risorsa che si desidera raggiungere.
- Il secondo parametro facoltativo (init) consente di personalizzare le opzioni della richiesta e di specificarne, ad esempio, gli headers o il body.

Non appena il server risponde, fetch() restituisce una Promise, indipendentemente dal codice di risposta HTTP (quindi restituisce una Promise indipendentemente al verificarsi o meno di un errore lato risorsa remota). Al suo interno la promise incapsula la risposta alla richiesta. Oltre a fornire un comodo strumento per raggiungere risorse remote, Fetch API offre funzioni per ottenere Promise che incapsulino i dati ottenuti convertiti in formati comuni come JSON (metodo json()) oppure plain-text (metodo text()).

```
// esempio di conversione della risorsa ottenuta
// in formato JSON
fetch("URL/TO/JSON")
.then( response => {
    // restituisco la promise
    // che incapsula la risorsa scaricata
    // convertita in formato JSON
    return response.json();
});

// esempio di conversione della risorsa ottenuta
// in formato Text
```

```
fetch("URL/TO/TEXT")
.then( response => {
    // restituisco la promise
    // che incapsula la risorsa scaricata
    // in formato plain-text
    return response.text();
});
```

Fetch restituisce delle Promise e pertanto, per ottenere l'effettivo valore della risorsa scaricata, si crea una catena di then():

```
fetch("URL/TO/JSON")
.then( response => {
    response.json().then( resource => {
        // resource contiene la risorsa
        // in formato JSON
        return resource;
    });
});

fetch("URL/TO/TEXT")
.then( response => {
    response.text().then( resource => {
        // resource contiene la risorsa
        // in formato plain-text
        return resource;
    });
});
```

Fetch API viene utilizzata nella classe AjaxService.

3.3 Ajax Service

La classe AjaxService (anch'essa scritta con TypeScript) fornisce un punto di accesso centralizzato per tutti quei servizi che richiedono l'interazione con delle sorgenti di dati remote e gestisce tutti gli aspetti connessi all'autenticazione, sessione e gestione degli errori in maniera trasparente agli utilizzatori esterni (ServiceIntegrator e ServiceStorage).

```
class AjaxService {
// effettua una richiesta asincrona all'url specificato come parametro
// possibile specificare se la richiesta deve essere autenticata
// e il formato dei dati restituiti
static async send( url:string, data: any = {}, authenticated = false,
format = 'text'){
    // se la richiesta al server remoto richiede autenticazione
    if( authenticated ) {
        // esecuzione delle operazioni per controllare che l'utente
        // sia correttamente autenticato (queste operazioni variano
        // in base ai meccanismi di autenticazione
        // che si ritengono piu' idonei)
    }
    try {
        // uso Fetch API per scaricare raggiungere la risorsa
        // remota tramite URL
        var response = await fetch(url, data).then( response => {
            // in caso di errore HTTP, lancio eccezione di errore
            // e non eseguo altro
            if( !response.ok ) {
                response.text().then(error => {
                    throw error;
                });
            }
```

```
            return response;
        }).then( data => {
            // in base al formato specificato come parametro
            // vengono richiamate i metodi json() o text()
            // della promise restituita dal fetch()
            switch ( format ){
                case 'text':
                    return data.text();
                case 'json':
                    return data.json();
            }
            throw ('[Ajax Service] Unsupported format!');
        });
        return response;
    } catch (error){
        throw ('[Ajax Service] '+error);
    }
}
}
```

3.4 Le interfacce EventHandler e Service

Sono due interfacce realizzate in TypeScript - la prima alla base del ServiceIntegrator e StorageService e la seconda per modellare i servizi connessi a ServiceIntegrator e StorageService - al fine di fornire allo sviluppatore uno scheletro per realizzare un event handler personalizzato o creare nuovi servizi. L'interfaccia Service prevede un unico metodo per ricordare allo sviluppatore che il servizio deve essere prima inizializzato. In questa fase deve registrare gli eventi e le relative callback nell'event handler designato (specificato nel parametro eventHandler):

```
// interfaccia che i nuovi servizi possono implementare
```

```
interface Service {
   init(eventHandler: EventHandler):void;
}
```

L'interfaccia EventHandler prevede tre metodi rispettivamente per ricordare allo sviluppatore che un event handler deve essere inizializzato, deve prevedere un metodo register per consentire ai servizi di registrarsi e un metodo emit per consentire agli utilizzatori di lanciare l'evento desiderato:

```
// interfaccia da implementare per realizzare nuovi servizi
// possono implementare
interface EventHandler {
    // metodo per inizializzare il nuovo event handler
    init(config: string): void;
    // metodo per registrare un evento nel event handler
    register(name:string, callback: Function):void;
    // metodo per lanciare un evento registrato
    emit(name:string, params: object, successCallback: Function, errorCallback: F
    serviceFactory(service:string):Service
    getServer():string;
}
```

3.5 Integrazione degli intermediari ServiceIntegrator e StorageService in LIME

Al fine di rispettare le specifiche di modularità richieste nella nuova versione di LIME sono state realizzate tre classi: ServiceIntegrator (sezione 3.5.1), StorageService (sezione 3.6), realizzate su modello di un event handler e AjaxService (sezione 3.7) come classe con metodo statico per effettuare richieste AJAX ad una risorsa remota.

3.5.1 Il ServiceIntegrator

Il ServiceIntegrator fornisce un'interfaccia comune per richiedere l'esecuzione di servizi ed è progettato per mascherarne la logica, in modo che possano essere spostati server-side o client-side - in base a criteri di efficienza, facilità di sviluppo e implementazione - in maniera trasparente al Message Passing Bus e alla vista. È progettato come un event handler seguendo l'approccio a callback, cioè una classe in grado di reperire la volontà da parte di classi esterne di registrare degli eventi sincroni o asincroni di attendere che questi vengano richiamati ed eseguire la corrispondente callback. Il seguente diagramma descrive il flusso delle operazioni alla ricezione in una richiesta di servizio da parte di un richiedente. In base all'evento richiamato, il ServiceIntegrator richiama un servizio locale (client-side) o un servizio remoto (server-side) tramite AjaxService. Infine, in caso di errore o successo della richiesta, restituisce il risultato al richiedente. Da questo diagramma si può dedurre come il richiedente (in questo caso il Message Passing Bus) sia ignaro della reale posizione del servizio richiesto. Solo il ServiceIntegrator ne conosce la reale posizione e in base a come è costruito il servizio, la richiesta viene inoltrata tramite AjaxService ad un server remoto, oppure risolta coinvolgendo un servizio client-side.

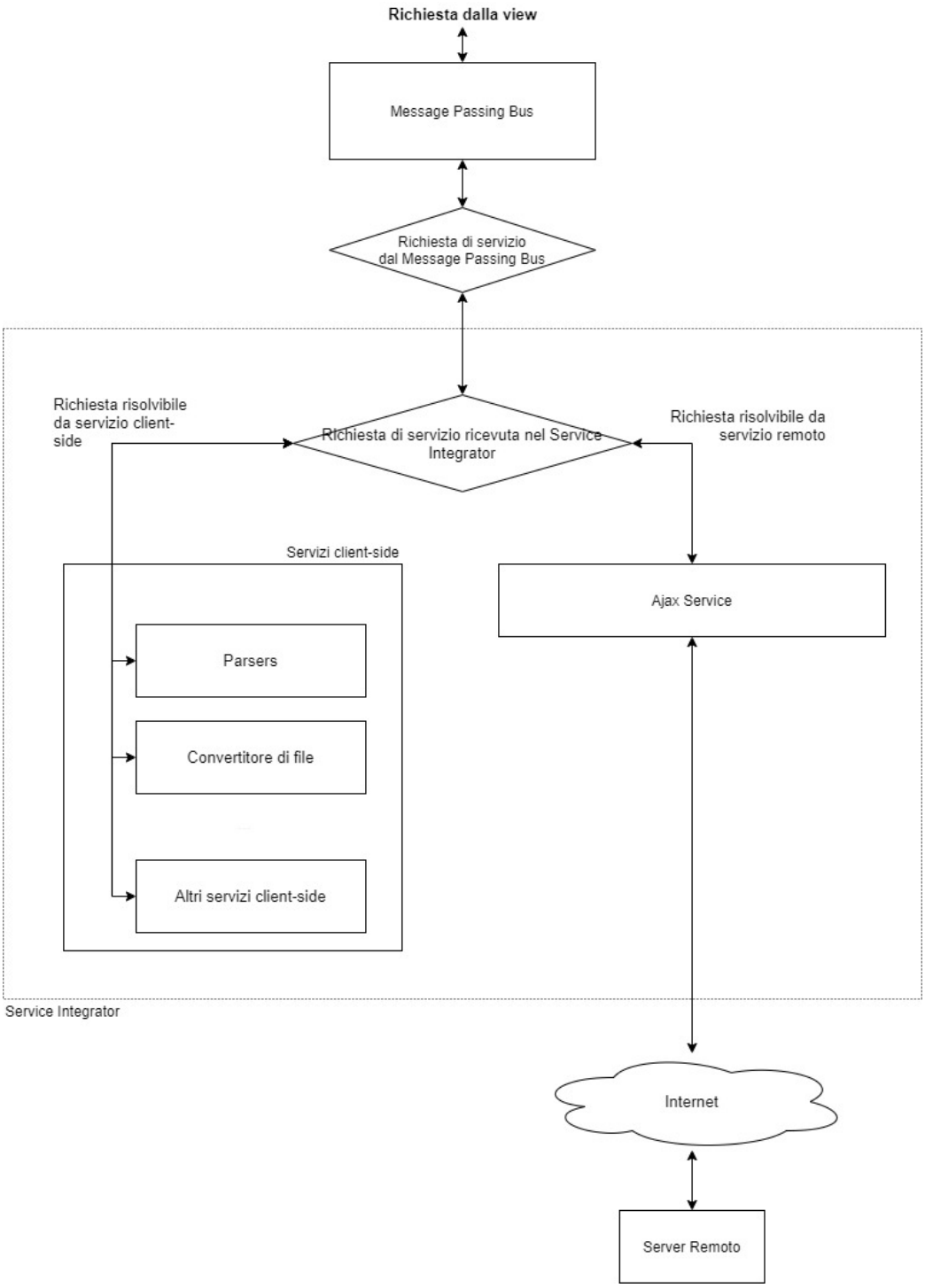


Diagramma di flusso di una richiesta al ServiceIntegrator

3.5.2 Servizi

I servizi costituiscono quella serie di operazioni che un'entità esterna (nel caso specifico di LIME il Message Passing Bus) può richiedere al ServiceIntegrator o al StorageService al fine di soddisfare una richiesta di un comando lanciato dall'utente o per soddisfare altre richieste provenienti dall'applicazione. Ciascun servizio è progettato come una classe TypeScript che implementa l'interfaccia ServiceInterface e dotata di metodi sincroni o asincroni corrispondenti alle funzioni callback relative agli eventi richiamabili nel servizio. Ciascuna funzione viene registata nell'event handler di riferimento come callback richiamata in seguito al lancio dell'evento. Di seguito un esempio di classe che modella un servizio:

```
// Esempio di implementazione di un nuovo servizio
class Service implements ServiceInterface {
    init( eventHandler: EventHandlerClass ){
        // registrazione delle callback nel event handler
        eventHandler.register("Service.sum", this.sumCallback);
        eventHandler.register("Service.getData", this.getDataCallback);
    }
    // ... qui sotto vengono dichiarate le callback registrate
    // nella funzione init
    sumCallback (params, eventHandler: EventHandlerClass): number {
        var op1: number = params.firstNum;
        var op2: number = params.secondNum;
        var result: number = op1 + op2;
        console.log( result ); // stampa il risultato nella console del browser
        return result;
    }
    async getDataCallback (params, eventHandler: EventHandlerClass): number {
        var url: string = params.url;
        try {
            var result:string = await AjaxService.send(url,
```

```
            {userID: params.userID});
        } catch ( error ) {
            throw new Error(error);
        }
    }
}
```

3.5.3 Implementazione del ServiceIntegrator

Il ServiceIntegrator è stato scritto utilizzando TypeScript, implementa l'interfaccia EventHandler (che definisce la struttura di un event handler generico) e non usa libreria esterne. È progettato come come classe TypeScript e riduce al minimo le dipenze software esterne. Dispone di tre funzioni pubbliche:

- init(config: string): per inizializzare il ServiceIntegrator.
- register(eventName: string, callback: function): che permette ad un servizio di registrare una sua operazione (evento) e la corrispondente callback che esegue l'operazione.
- emit (eventName: string, params: any, successCallback: function, errorCallback: function): invocata dal richiedente (nello specifico il Message Passing Bus) per lanciare un evento e richiedere esecuzione di un servizio.

Di seguito una panoramica della classe:

```
class ServiceIntegrator implements EventHandler {
    // ... alcuni campi della classe ...
    init(config:string){
        // parsing della configurazione per ottenere lista
        // servizi attivi per l'istanza della web app
        // e URL al server dell'applicazione
```

```
    }
    register(eventName:string, callback: function){
        // registrazione degli eventi nella struttura dati preposta
        // ad ogni nome dell'evento corrisponde una callback
    }
    emit(eventName: string, params: any,
    successCallback: function, errorCallback: function) {
        // creazione della Promise di esecuzione dell'evento
        // ed esecuzione della callback di successo
        // o della callback di insuccesso
    }
}
```

Si illustrano ora nel dettaglio le singole funzioni.

3.5.4 Metodo init()

Il metodo init esegue le operazioni di inizializzazione del ServiceIntegrator, preparando l'insieme dei servizi eseguibili (nello specifico dal Message Passing Bus) ed inserendo gli eventi registrati in una apposita struttura dati. In questa prima implementazione, la struttura dati individuata è l'array. Il metodo init() viene richiamato esternamente per modificare, in qualsiasi momento, la configurazione del ServiceIntegrator senza dover istanziare un nuovo oggetto.

```
init( config: string ){
    // parsing della stringa che contiene ogetto
    // JSON di configurazione della web app
    this.config = JSON.parse(config);
    // indirizzo del server della web application
    this.server = this.config['server'];
    // array dei servizi disponibili per l'istanza dell'applicazione
    this.enabledServices = this.config['services'];
```

```
    // per ogni servizio configurato
    var instance = this;
    enabledServices.forEach(service => {
        // si ottiene l'oggetto del servizio dal metodo Factory
        var serviceObject = instance.serviceFactory(service);
        // si esegue la init per registrare gli eventi del servizio
        svcObj.init(instance);
    });
}
```

L'unico parametro di questo metodo è di tipo string e contiene l'oggetto in formato JSON con i dati necessari alla configurazione del ServiceIntegrator. Di seguito un esempio:

```
{
    "server": "URL/to/server",
    "services" ["Service1", "Service2"]
}
```

Dove "services" è un array di string ciascuna pari al nome della classe del servizio che si desidera abilitare nell'event handler appena inizializzato.

3.5.5 Metodo register()

Il metodo register offre ai servizi un punto di accesso per la registrazione degli eventi. Al suo interno vengono eseguite le operazioni per inserire l'evento registrato e la relativa funzione callback all'interno dell'apposita struttura dati che memorizza l'insieme degli eventi registrati. Per questa implementazione si è scelto un array, la cui chiave corrisponde al nome dell'evento e il valore corrispondente è la callback relativa a quell'evento.

```
register( eventName:string, callback:function){
    console.log("Registering event: "+name);
    // Inserisce nell'array degli eventi
```

```
    // la rispettiva callback
    this.enabledServices[eventName] = callback;
}
```

3.5.6 Metodo emit()

Il metodo emit() viene richiamato quando si desidera lanciare un evento registrato ed eseguire l'operazione di un servizio. In concreto, emit() ricerca per chiave l'evento specificato dal primo parametro all'interno della struttura dati (array) preposta alla memorizzazione degli eventi e richiama la callback corrispondente alla chiave nome evento richiesto. Successivamente incapsula l'esecuzione all'interno di una Promise, gestendo in modo trasparente eventuali operazioni asincrone o sincrone. In questo modo, il Message Passing Bus o più in generale l'entità esterna, non deve necessariamente determinare se l'evento che sta chiamando è asincrono o sincrono. Si deve semplicemente occupare di prevedere due diverse callback, una per i casi di successo e una per i casi di insuccesso e a loro interno gestire i due casi come si ritiene più opportuno.

```
// primo parametro corrisponde al nome dell'evento, il secondo
// i dati necessari all'evento e i successivi sono rispettivamente
// la callback di successo e insuccesso
emit( eventName:string, params: any, successCallback: function,
errorCallback: function){
    var instance = this;
    var promise = new Promise(function (resolve) {
        // controlla se l'evento desiderato esiste
        if( instance.enabledServices[eventName] !== undefined ){
            // l'evento desiderato esiste lancia della callback corrispondente
            return resolve(instance.enabledServices[name](params, instance));
        }
        // se l'evento non \`e stato registrato lancia eccezione
        throw new Error("Event "+eventName+" has not yet been registered.");
```

```
    });
    promise.then( result => {
        // Promessa soddisfatta
        // esecuzione callback di successo
        // passando il risultato
        successCallback(result);
    })
    .catch( error => {
        // cattura degli errori e delle eccezioni
        // esecuzione della callback di errore
        // passando errore come parametro
        errorCallback(error);
    });
}
```

3.6 Il StorageService

La classe StorageService (anch'essa scritta con TypeScript) ha l'obiettivo di gestire l'apertura dei file, salvarne il contenuto, duplicarli e spostarli, importare ed esportare i documenti e se necessario convertirli in altri formati. Interagisce con lo storage server-side, ma è progettata per implementare in futuro il salvataggio del documento corrente nel local storage del browser, per far fronte ad eventuali impossibilità di salvataggio del documento o per consentire la modifica del documento off-line. Il tutto in modo trasparente al richiedente. La classe StorageService e ServiceIntegrator sono simili. Infatti, anche il StorageService è progettato come event handler e l'insieme delle operazioni che i servizi offrono al richiedente sono progettate sottoforma di eventi, ai quali corrispondono delle callback. Non viene illustrata nel dettaglio della classe, in quanto ServiceIntegrator e StorageService sono praticamente sovrapponibili. StorageService e ServiceIntegrator sono distinti come due diverse classi e oggetti e pertanto non condividono la stes-

sa lista di eventi registrati. Infatti, nel StorageService vengono registrati esclusivamente servizi che riguardano la gestione dei documenti, mentre nel ServiceIntegrator vengono registrati servizi che riguardando il parsing e altro genere di servizio.

3.7 I moduli ServiceIntegrator, StorageService e AjaxService

Dalla versione ES6, JavaScript ha introdotto il concetto di moduli e tale concetto è stato adottato anche da TypeScript. I moduli sono un insieme di classi, funzioni o variabili eseguiti nel proprio scope e non in quello globale. Quindi significa che, affinché lo scope sia visibile da altre classi e funzioni, deve essere esplicitamente esportato (keyword export) ed importato in un altro modulo (keyword import). Di seguito un esempio. Considerando un qualsiasi file Classe1.ts:

```
// si esporta lo scope della classe Classe1
// per rendere visibile lo scope
// anche in altre classi
export Classe1 {
    // costrutture, campi e metodi della classe...
}
```

Per importare lo scope della Classe1 ed istanziare un oggetto della classe è necessario importare lo scope tramite la keyword import. Si supponga di avere un differente file main.ts:

```
import {Classe1} from "path/to/Classe1.ts"
// senza import del modulo non sarebbe possibile istanziare oggetto di Classe1
var classe1 = new Classe1();
```

Seguendo questo approccio, ServiceIntegrator e StorageService sono stati ideati come moduli e importano tutte le classi di cui hanno bisogno per fun-

zionare (quindi classi dei servizi e AjaxService). Sono facilmente integrabili in un preesistente progetto importando le relative classi:

```
import {ServiceIntegrator} from "path/to/ServiceIntegrator.ts"
import {StorageService} from "path/to/StorageService.ts"
```

A seguito dell'importazione sarà possibile istanziare oggetti delle due classi ed inizializzare i due event handler con la lista dei servizi attivi. Segue un esempio:

```
// si importa la classe ServiceIntegrator
import {ServiceIntegrator} from "path/to/ServiceIntegrator.ts"
// viene istanziato un nuovo oggetto di classe ServiceIntegrator
var ServiceIntegrator = new ServiceIntegrator();
serviceIntegrator.init("config string");
// viene richiesto lancio di un evento
serviceIntegrator.emit("Service.eventName", params,
successCallback(){
    // operazioni in caso di successo
}, errorCallback(){
    // operazioni in caso di errore
} );
```

3.7.1 Creare un nuovo event handler

ServiceIntegrator e StorageService non sono gli unici event handler utilizzabili dallo sviluppatore. Infatti, è possibile creare un nuovo event handler implementando l'interfaccia EventHandler e importando le interfacce EventHandler e Service (contenute nel file Interfaces.ts). Di seguito un'esempio di come creare un nuovo event handler:

```
import {EventHandler} from "path/to/Interfaces.ts";
// nuovo event hadler
export class NewEventHandler implements EventHandler {
```

```
    // array degli eventi registrati
    private enabledServices = string[];
    init( config:string ){
        // svolge operazioni di inizializzazione
    }
    register( eventName:string, callback: function ){
        // registrazione degli eventi nella struttura dati preposta
    }
    emit (eventName: string, params: any,
    successCallback: function, errorCallback: function){
        // ricerca della callback relativa all'evento specificato
        // nella struttura dati preposta e successiva esecuzione
        // della callback trovata
    }
}
```

3.7.2 Aggiungere nuovi servizi a ServiceIntegrator o StorageService

In questa sezione si spiega come aggiungere nuovi servizi al ServiceIntegrator (in egual modo anche al StorageService). Dopo aver creato il nuovo servizio ed aver esportato la classe che lo realizza:

```
    import {Service, EventHandler} from "path/to/Interfaces.ts"

    export class NewService implements Service {
        init(eventHandler: EventHandler){
            eventHandler.register("NewService.newEvent", this.newEvent);
        }
        async newEvent(params:any, eventHandler; EventHandler) {
            // svolge qualche operazione e restituisce un risultato
        }
```

```
    }
```

Sarà sufficiente importare la classe nel file che realizza il ServiceIntegrator ed aggiungere la classe al set di classi previste nel metodo serviceFactory():

```
import {EventHandler} from "path/to/Interfaces.ts"
import {NewService} from "path/to/NewService.ts"

export class NewEventHandler implements EventHandler {
    // array degli eventi registrati
    private enabledServices = string[];
    init( config:string ){...}
    register( eventName:string, callback: function ){...}
    emit (eventName: string, params: any, successCallback: function, errorCallba
    // si aggiunge NewService all'interno del metodo serviceFactory
    serviceFactory(service:string){
        const services: {
            .. altre classi di servizi
            NewService,
            ... altre classi di servizi
        }
    }
}
```

Capitolo 4

Valutazione

In questo capitolo si cerca di dare una valutazione qualitativa sui miglioramenti apportati dall'utilizzo della libreria in termini di facilità di sviluppo, proponendo un confronto tra il vecchio approccio di creazione dei componenti di LIME e il nuovo introdotto con la libreria. Come indicato nei capitoli precendenti l'attuale versione di LIME è costituita da una serie componenenti (detti "package") i quali realizzano la logica di servizio utilizzando le API messe a disposizione da ExtJS e gestiscono alcuni settaggi della view. Si può ben capire, da questo incipit, che il codice dei componenti di LIME è composto da codice che si occupa di cose diverse. Prima di proseguire viene illustrato un esempio di come veniva creato un componente ed un esempio di alcune funzioni che interagiscono con risorse remote. Si supponga di analizzare il componente "Server" di LIME che incapsula le chiamate REST al server e di analizzare nello specifico la funzione di salvataggio di un file e la funzione di esportazione di un file:

```
Ext.define('LIME.Server', {
    // .. altri campi e funzioni
    // Register user.
    register: function (user, success, failure) {
        this.request({
            method: 'POST',
```

```
            url: '{nodeServer}/documentsdb/Users',
            jsonData: user,
            success: success,
            failure: failure
        });
    },
    saveDocument: function (path, content, success, failure) {
        this.authRequest({
            method: 'PUT',
            rawData: content,
            url: '{nodeServer}/documentsdb/Documents' + path,
            success: function (response) {
                console.info('Saved', path);
                success(response.responseText);
            },
            failure: function (error) {
                console.warn('Saving document failed:', error);
                if (failure) failure(error);
            }
        });
    },
    // Export a document to a url accessible to everyone
    exportDocument: function (path, success, failure) {
        var me = this;
        this.authRequest({
            method: 'POST',
            url: '{nodeServer}/documentsdb/Export?url=' + path,
            success: function (response) {
                var url;
                try {
                    url = JSON.parse(response.responseText).url;
```

```
                    url = me.getNodeServer() +
                    url.substring(url.indexOf('/documentsdb/'));
                } catch (e) {
                    console.warn('Error exporting file');
                    console.warn(response.responseText);
                    if (failure) failure();
                }
                if (url) success(url);
            },
            failure: failure || function (error) {
                console.warn('Error exporting file', error);
            }
        });
    },
    // ... altri campi e funzioni
});
```

Si notino alcune criticità:

- La gestione degi casi di successo o insuccesso avviene tramite due callback e non risulta chiaro quali operazioni svolgano queste callback;
- Le operazioni di parsing vengono svolte all'esterno della chiamata AJAX costringendo lo sviluppatore a ripetere righe di codice per effettuare operazioni ridondanti.

L'approccio proposto dalla nuova libreria per la creazione di un servizio si appoggia sulle direttive di scrittura di una classe TypeScript e istanziazione di un oggetto. Concetti che uno sviluppatore dovrebbe conoscere bene. Inoltre, la possibilità di scrivere la logica dei servizi utilizzando le keyword async/await fa si che, a parità di numero di righe di codice, esso risulti più lineare e facile da leggere. Inoltre, AjaxService risolve implicitamente problemi di autenticazione e gestione degli errori evitando più possibile codice ridondante. Segue un esempio:

```
export class DocumentService implements Service {
    // ... altri campi e metodi
    async saveFile( params, storageService: StorageService){
        // check if path is set
        if(params.path == undefined){
            throw "No path set";
        }
        // preparing url
        var urlRequest = '{nodeServer}/documentsdb/Documents/{path}';
        urlRequest = urlRequest.replace('{nodeServer}', storageService.getServer
        urlRequest = urlRequest.replace('{path}', params.path);
        // preparing header
        var data = {
            method: 'PUT',
            body: params.content
        }
        var response = await AjaxService.send(urlRequest, data, true, 'text');
        return response;
    }
    // ... altri campi e metodi
}
```

Il controllo dei parametri, la preparazione dell'URL, la preparazione dei dati per la richeista e la chiamata effettiva, sono scritti in maniera lineare ed è chiaro allo svilupattore dove intervenire in caso di problemi.

Obiettivo futuro è quello di sostituire la Promise incapsulata nel metodo emit() di ServiceIntegrator e StorageService con le keywords async e await. Sono due keywords da utilizzare nel contesto di funzioni o metodi asincroni. Nello specifico, async è una keyword che deve essere inserita nella firma del metodo da categorizzare come asincrono e implicitamente configura un approccio promise-based senza dover esplicitamente utilizzare l'oggetto Promise [6]. Si chiarisce il concetto, comparando due casi; il primo utilizzando

l'oggetto Promise e il secondo usando le keywords async/await.

```
// function asincrona
function doSomethingAsync(){
    // fa qualcosa di asincrono
}
```

Si richiama la funzione doSomethingAsync() utilizzando le Promise:

```
var promise = new Promise(function(resolve, reject){
    var result = doSomethingAsync();
    if (result) {
        resolve(result);
    } else {
        reject(result);
    }
});
// si ottiene il risultato della promise
// con la catena di then()
promise.then (result => {
    // operazioni svolte con result
})
```

Il codice che si ottiene è decisamente più caotico. Utilizzando le keyword async/await, si ottiene una maggiore pulizia del codice:

```
// dichiarazione della funzione come asincrona
// usando la keyword async
async function doSomethingAsync(){
    // fa qualcosa di asincrono
}
```

Si richiama la funzione doSomethingAsync() anteponendo alla chiamata della funzione, la keyword await:

```
var result = await doSomethingAsync();
result.then( result => {
    // operazioni con il risultato
})
```

Durante la fase di inizializzazione (metodo init()), una classe che modella un servizio, registra il set degli eventi nel event handler corrispondente tramite il metodo register(). Tale metodo prende come primo parametro il nome dell'evento che è una stringa che lo identifica, ma su quella stringa non viene effettuato alcun controllo. Questo comporta possibili conflitti nei nomi. Infatti, due servizi eventualmente omonimi possono sovrascrivere l'evento registrato dall'altro. Per risolvere questo problema sarebbe sufficiente inserire nella funzione register un ciclo che controlla se per il nome dell'evento esiste già un callback associata. Se esiste, si lancia un'eccezione e il nuovo evento non viene registrato.

Capitolo 5

Conclusioni

La possibilità di disaccoppiare due componenti di una applicazione, introducendo nuovi livelli intermedi nella sua architettura, permette di raggiungere gradi di modularità tali da rendere il software facilmente scalabile e adattabile ai cambiamenti repentini. Infatti, l'indebolimento del legame tra due componenenti di un software, consente di sostituirne uno senza che la controparte ne risenta. Questo rappresenta un grande punto di forza per un'applicazione rendendola resiliente. L'introduzione di nuovi livelli, d'altro canto, introduce due problemi tipici di queste architetture:

- Aumento di complessità che cresce di pari passo con il numero e la complessità dei livelli intermediari aggiunti.
- Diminuzione delle performance di un'applicazione a causa del coinvolgimento di più livelli nell'esecuzione di una singola richiesta.

In casi, come LIME, la possibilità di sostituire componenti dell'applicazione intervenendo soltanto in una più piccola porzione di codice riduce notevolemtne i tempi di sviluppo e supera di gran lunga gli svantaggi tipici di un'architettura a livelli. Inoltre, vista la natura dei dati trattati dall'applicazione, eventuali cali di performance non sono percettibili dall'utente. La libreria è realizzata in modo da poter essere adattata in pochi passi ed in-

tegrata nel progetto preesistente oppure offre la possibilità di implementare l'interfaccia EventHandler e realizzare un event handler personalizzato:

- Nel primo caso, usando le classi ServiceIntegrator e StorageService che realizzano e soddisfano due esigenze comuni nelle applicazioni web (il primo per servizi generici e l'altro specifico per lo storage);
- Nel secondo caso, implementando l'interfaccia EventHandler, lo sviluppatore può creare un nuovo event handler personalizzato.

Il secondo caso costituisce una possibilità di realizzare quanti event handler si desidera e come li si desidera. Infatti, implementando l'interfaccia EventHandler lo sviluppatore ha a disposizione uno scheletro che è obbligato ad implementare per realizzare i metodi fondamentali di un event handler, lasciando comunque la libertà di scelta della struttura dati nella quale memorizzare gli eventi (ad esempio, come array o come map) oppure se per la risoluzione dei servizi sia necessario avere un server remoto. Nulla vieta allo sviluppatore di registrare nell'event handler solo eventi sincroni e locali che non necessitano di risorse remote, mantentendo il paradigma di disaccoppiamento tra le parti offerto da questa architettura.

Bibliografia

[BDD18] Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Stephan T. Larsen, Manuel Mazzara "From Monolithic to Microservices: An Experience Report from the Banking Domain" 2018 https://ieeexplore.ieee.org/abstract/document/8354415

[DEL19] Lorenzo De Laurentis "From Monolithic Architecture to Microservices Architecture" 2019 https://ieeexplore.ieee.org/document/8990350/

[PMA16] Francisco Ponce, Gastòn Márquez, Hernán Astudillo "Migrating from monolithic architecture to microservices: a Rapid Review" 2016 https://ieeexplore.ieee.org/document/8966423/

[CLC05] Ivica Crnkovic, Stig Larsson and Michel Chaudron "Component-based Development Process and Component Lifecycle" 2005 https://hrcak.srce.hr/index.php?id_clanak_jezik=69311&show=clanak

[1] Component-Based Architecture - https://www.tutorialspoint.com/software_architecture_design/component_based_architecture.htm - Ultima visita: 1/7/21

[2] *Component-Based Architecture* - https://www.tutorialride.com/software-architecture-and-design/component-based-architecture.htm - Ultima visita: 1/7/21

[3] *LIME and System Architecture* - http://lime.cirsfid.unibo.it/ - Ultima visita: 1/7/21

[VP21] Simplex Design Plan - Fabio Vitali, Monica Palmirani `https://docs.google.com/document/d/14fMz7-m2JCOoZh3fjVMb08Dele8czqhLDNHozCG_OtA/edit`

[4] *Javascript* - MDN Web Docs `https://developer.mozilla.org/en-US/docs/Web/JavaScript?retiredLocale=it` - Ultima visita 23/6/21

[5] *Promise* - MDN Web Docs - `https://developer.mozilla.org/` - Ultima visita 23/6/21

[6] *Async/Await* - MDN Web Docs - `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function` - Ultima visita 23/6/21

Ringraziamenti

Un sentito ringraziamento a parenti e amici. In particolare ai miei genitori Nadia e Angelo, a mia sorella Federica ai miei nonni Piera, Michele, Marilena, Carlo, per il sostegno datomi durante questo percorso.