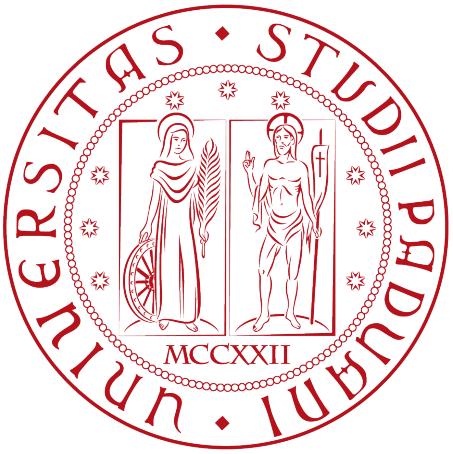
Università degli Studi di Padova

Dipartimento di Matematica

Corso di Laurea in Informatica



Editor visuale per la manipolazione di template HTML

Tesi di laurea triennale

Relatore

Prof. Claudio Enrico Palazzi

Laureando

Daniele Marin

Anno Accademico 2016-2017

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di trecentoventi ore, dal laureando Daniele Marin presso l'azienda Zucchetti S.p.a.. L'obiettivo di tale attività di stage è l'analisi di varie librerie Javascript per la realizzazione di template HTML, al fine di poter realizzare un editor grafico che permetta la selezione e la modifica dei template per un loro sucessivo inserimento all'interno di pagine HTML. Inoltre è stato effettuato uno studio sul comportamento dei template in ambito responsive, sulla possibilità di inserire plug-in jQuery all'interno dei template e su di un metodo di caricamento delle librerie controllato in modo di non avere più istanze della stessa libreria se utilizzata da diversi template.

Ringraziamenti

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei frammenti di codice

Capitolo 1

Introduzione

1.1 L'azienda

Figura 1.1: Logo di Zucchetti S.p.a.

La Zucchetti S.p.a. è una software house con sede a Lodi, che si occupa di soluzioni complete per le aziende, professionisti(commercialisti, consulenti del lavoro, avvocati, curatori fallimentari, notai ecc.) e pubbliche amministrazioni(Comuni, Province, Regioni, Ministeri, società pubbliche ecc.).
Il gruppo Zucchetti è la prima azienda italiana in Europa con oltre 3300 addetti, 1100 partner e oltre 105000 clienti.
Le soluzioni principali proposte dall'azienda sono :

- Softwere: gestionali, per la sicurezza sul lavoro, analisi business ecc.
- Hardware: per la rilevazione presenze, controllo accessi e controllo produzione.
- Servizi: di outsourcing, cloud computing e data center.

1.1.1 Portal Studio

Lo stage si è svolto nella sede distaccata di Padova che si occupa di ricerca e sviluppo. Tra i software che vengono sviluppati in questa sede è presente Portal Studio che consiste in una WEB application per la creazione di siti web.
L'applicazione offre all'utente un set completo di strumenti per la creazione di pagine web, permette la creazione e modifica in modo grafico della struttura HTML, la gestione degli stili tramite editor grafico per il CSS ed inoltre permette di gestire i dati provenienti da diversi tipi di database, il loro filtraggio e il binding con varie strutture HTML come liste e tabelle.
Il software risulta essere molto maturo e oltre alle funzionalità sopracitate permette

anche la creazione di portlet e pagelet e altri elementi riutilizzabili e la gestione di risorse come dati in formato JSON.

1.2 Il progetto

Il progetto proposto dall'azienda per lo stage, nasce dal desiderio di aggiungere all'applicazione Portal Studio una nuova funzionalità che consiste nell'offrire all'utente la possibilità di inserire nelle proprie pagine HTML dei template già pronti e selezionabili da un insieme prestabilito.
Questo desiderio ha portato l'azienda ad interessarsi ai template engine come Mustache.js, HandlebarJS ecc.
Lo stage si divide in due parti.
La prima consisteva nello studio dei template e degli aspetti ad essi correlati, la seconda nella realizzazione di un editor che ne permettesse la visualizzazione e la modifica.
Le parti più rilevanti del lavoro svolto sono state sicuramente:

- la scelta della libreria da utilizzare nello sviluppo del progetto, che è stata individuata analizzando le caratteristiche di varie librerie per il templateing, cercando fra queste quella che in miglior modo si adeguasse ai requisiti del progetto. Questa libreria è stata scelta sapendo che l'azienda l'avrebbe utilizzata per il proprio applicativo quindi i risultati dell'analisi effettuata sono stati discussi con il tutor aziendale che si occupa inoltre del reparto di ricerca e sviluppo;
- l'integrazione di plug-in JQuery all'interno del file dove è definito il template;
- la realizzazione della parte dell'editor relativa alla modifica dei dati, che permette di adattarsi a qualsiasi tipo di template HTML ed SVG e inoltre, nonostante i pochi tipi primitivi forniti da JavaScript, di individuare a quale elemento (immagine, URL, colore, ecc.) si riferiscano i dati dei template.

1.2.1 Prima parte

La prima parte del progetto inizia con la realizzazione di qualche template prototipo, utile sia per studiare le possibilità della libreria scelta sia per avere un insieme di template da inserire nell'editor che è stato realizzato in seguito.
Durante questa parte del progetto l'attenzione è stata rivolta alla possibilità di realizzare template statici, dinamici, template come composizione di altri template (es. lista di contatti) e template che utilizzano SVG.
In seguito alla realizzazione dei template prototipo è stato effettuato uno studio sulla possibilità di rendere i template responsive cioè permetterne la visualizzazione sia su dispositivi desktop che mobile.
La prima parte si è conclusa con uno studio sulla possibilità di realizzare template che contenessero al loro interno plug-in JQuery e sulla gestione del caricamento delle librerie necessarie al funzionamento dei template all'interno della pagina HTML.

1.2.2 Seconda parte

La seconda parte del progetto consisteva nella realizzazione di un editor grafico che permette all'utente la selezione di un template da una lista prestabilita. In seguito alla selezione del template desiderato quest'ultimo verrà visualizzato in un box dedicato e tramite un editor, che viene costruito sulla base dei dati editabili del template (i dati

sono contenuti in un oggetto JSON che fa parte del template), è possibile vedere il comportamento del template durante la modifica dei suoi dati.
Per determinati template, come quelli considerati composti, l'editor deve dare la possibilità di visualizzare direttamente l'oggetto JSON contenente i dati e permetterne la modifica.
Non essendo presente una struttura di beck-end proposta dall'applicazione Portal Studio, perché ancora in fase di valutazione, l'editor è stato sviluppato separatamente, l'insieme dei template consiste in un gerarchia di directory contenenti i vari elementi che compongono i template (file HTML, JSON, immagini e librerie) suddivise per categoria.
Il caricamento delle risorse viene eseguito tramite chiamate http-get request, non essendo presenti delle API fornite dall'azienda.

Capitolo 2

Librerie analizzate

In questo capitolo vengono messe a confronto varie librerie *JavaScript* che permettono la realizzazione di template HTML, ne vengono analizzati i pregi e i difetti per arrivare a descrivere i motivi che hanno portato alla scelta della libreria utilizzata nel progetto.

2.1 Considerazioni generali

Negli ultimi anni sono nate molte librerie che permettono la creazione di template HTML che hanno portato notevoli vantaggi agli sviluppatori, offrendo loro un nuovo strumento che permette di creare modelli HTML per la rappresentazione dei dati e riutilizzarli all'interno di pagine differenti con una considerevole diminuzione del codice JavaScript e HTML che normalmente viene utilizzato per la modifica de DOM. Queste librerie si sono evolute velocemente fino ad arrivare a permettere agli sviluppatori di creare intere User interface per applicazioni web, creare componenti riutilizzabili ed in qualche caso offrire funzionalità avanzate come il two-way binding.

2.1.1 I template con sintassi mustache

Le librerie studiate durante lo stage utilizzano tutte questa particolare sintassi, che permette di rappresentare variabili, sezioni, parziali ed altri elementi utili alla creazione del template, tramite l'inserimento di **tag**.
Questi particolari **tag** sono caratterizzati dall'utilizzo delle parentesi graffe come delimitatori e questo è il motivo per cui vengono definiti mustaches (baffi in inglese).
I tag si presentano nella forma "{{I P}}" dove I è un simbolo o una stringa ed identifica il tipo di tag, mentre P è un parametro o una chiave appartenente all'oggetto JSON correlato al template.
Per l'inserimento di variabili o parziali il **tag** è singolo, mentre per l'inserimento di sezioni, controlli del tipo not-exist ed altri, sono presenti un **tag** di apertura ed uno di chiusura.
Per capire meglio il funzionamento che sta alla base di questi template engine è utile fare degli esempi.

Questo esempio mostra il rendering di due variabili.

```
// oggetto JSON contenente i dati
var dati = { name: "Jon", age: 35};
// template HTML con l'aggiunta del tag mustache
var template = "<h1>Il mio nome è {{name}} e ho {{age}} anni.</h1>";

// il risultato del rendering sarà:

Il mio nome è Jon e ho 35 anni.
```

Codice 2.1: Esempio di template rappresentante una variabile.

In questo esempio viene definito il template per rappresentare una lista di prodotti.

```
// oggetto JSON contenente i dati
var dati = prodotti: [
                { name: "pane" },
                { name: "pasta" },
                { name: "biscotti" }
            ];

// template HTML con l'aggiunta del tag mustache
var template = "<p>Lista della spesa:
          <ul>
            {{#prodotti}}
              <li>{{name}}</li>
            {{/prodotti}}
          </ul>
        </p>";

// il risultato del rendering sarà:

Lista della spesa:
- pane
- pasta
- biscotti
```

Codice 2.2: Esempio di template rappresentante una sezione.

2.2 Mustache.js

Figura 2.1: Logo Mustache

Mustache può essere considerato come il padre dei template system, è open-souce e logic-less e presenta implementazioni per i più famosi linguaggi di programmaione, come Java, Phyton, Ruby, PHP, JavaScript e molti altri.
Mustache.js è un implementazione per JavaScript del template system Mustache.
La libreria è molto leggera e versatile visto che permette il rendering sia lato server che lato client.
Le funzioni offerte sono *render* e *parse*, la prima si occupano di creare la stringa HTML contenente il template renderizzato partendo dai dati JSON e dal template HTML e la seconda è opzionale e permette di preparare il template in modo da velocizzare l'operazione di render.
Mustache.js viene definita logic-less perché non presenta nessun tipo di costrutto *if-then-else* e loop come *for* o *do-while*

2.2.1 Come funziona

Il suo funzionamento è molto semplice dato che la libreria offre solamente due metodi.
Il metodo principale è *Mustache.render(data, template)* che prendendo in ingresso il template HTML, con gli opportuni **tag** mustache, e i dati tramite oggetto JSON, restituisce una stringa risultato dall'interpolazione dei due elementi.
La stringa risultante deve essere inserita all'interno della pagina tramite manipolazione del DOM.
Sfortunatamente Mustache.js non offre strumenti per la manipolazione del DOM per cui l'inserimento dovrà essere fatto dal programmatore utilizzando funzioni offerte dallo standard JavaScript o da altre librerie come JQuery.

2.2.2 Pregi e difetti

Uno dei pregi principali è sicuramente la semplicità e la leggerezza della libreria.
Inoltre la stesura dei template risulta intuitiva e semplificata dall'assenza di costrutti *if-else* o cicli *for* .
Come contro si può citare l'impossibilità di creare funzioni aggiuntive per la gestione dei template, possibilità offerta da altre librerie e la totale mancanza di strumenti per la manipolazione del DOM e dei dati del template che costringe il programmatore ad appoggiarsi ad altre librerie.

I template una volta renderizzati sono statici e un cambiamento nei dati non ha effetto sulla loro rappresentazione.

2.3 HandlebarsJS

Figura 2.2: Logo HandlebarsJS

HandlebarsJS è una libreria costruita sopra a Mustache quindi offre tutte le funzionalità di quest'ultimo e aggiunge al normale set di tag anche costrutti di controllo come l'espressione *if* e iteratori come *each*.
La libreria offre anche un set di metodi globali che permettono al programmatore di effettuare varie operazioni sul template e i suoi dati, in più la possibilità di creare delle funzioni personalizzate che possono essere inserite in uno spazio globale e riutilizzate a piacere su diversi template.
Handlebare permette di precompilare il template e offre prestazioni migliori rispetto a Mustache.

2.3.1 Come funziona

Il funzionamento di HandlebarsJS è simile a quello di Mustache, avendo a disposizione l'oggetto JSON contenente i dati e il template, prima si compila il template tramite il metodo *Handlebars.compile(template)*, il risultato della compilazione è una funzione che richiamata passandole come parametro l'oggetto JSON provvede ad interpolare i dati con il template e restituisce una stringa HTML che dovrà essere inserita nella pagina.
Anche in questo caso l'inserimento del template nella pagina deve essere fatto utilizzando strumenti esterni alla libreria perché essa non offre funzioni adeguate.

2.3.2 Pregi e difetti

Anche per Handlebars la leggerezza della libreria e la velocità nel rendering è da considerare un pregio.
Inoltre la possibilità di sfruttare un set di metodi e di poterne creare di propri risulta un vantaggio rispetto a Mustache.
Come Mustache i template una volta creati sono statici e una modifica dei dati non causa un aggiornamento del template che deve essere nuovamente ricompilato.

2.4 Ractive.js

Figura 2.3: Logo Ractive.js

Ractive.js è una libreria sviluppata al theguardian.com per creare **reactive interface** per il web, in modo semplice e efficente.
Questa libreria utilizza la sintassi mustache al pari delle librerie viste in precedenza, quindi un template scritto per funzionare con Mustache.JS o HandlebarsJS può essere utilizzato anche con Ractive.js.
Ractive.js arricchisce la sintassi mustache con nuovi strumenti come:

- Array index;
- Object iterator;
- Special e restricted reference;
- Espressioni;
- Alias.

Oltre all'estensione della sintassi mustache, Ractive.js offre un set molto variegato di opzioni e metodi che possono essere utilizzati per modificare sia il DOM del template che i suoi dati, per l'emissione e la ricezione di segnali e la gestione di animazioni.
Tutti questi strumenti permettono la realizzazione di template interattivi e di una certa complessità, cosa che non è possibile fare utilizzando le librerie precedentemente descritte.
La libreria è compatibile con tutti i browsers e offre un'ottima compatibilità con SVG e lo standard ES15.
I template ottenuti con Ractive.js offrono un *binding* tra la rappresentazione del template e il data model, quando i dati vengono modificati la libreria modifica in modo intelligente ed efficente il DOM del tempalte.

2.4.1 Come funziona

Ractive.js utilizza un approccio differente rispetto alle librerie viste in precedenza.
In particolare viene istanziato un oggetto di tipo Ractive che può essere inizializzato implementando varie opzioni tra quelle offerte dalla libreria.
Le opzioni principali sono l'oggetto JSON contenente i dati, il template e l'elemento HTML a cui dovrà essere agganciato il template all'interno del DOM.

Una volta istanziato l'oggetto viene creata nella **RAM** una rappresentazione virtuale del DOM (virtual DOM) e viene popolato il model con i dati contenuti nel JSON.
In seguito viene renderizzata nel browser la rappresentazione del template interpolato con i dati ad esso relativi.
Ia libreria si occupa di creare un legame tra il model e il virtual DOM.
Nel momento in cui il model subisce variazioni viene effettuato un confronto tra DOM virtuali e nel caso in cui venga rilevato un cambiamento la libreria si occupa di modificare il DOM reale in modo intelligente, cioè andando a modificare solamente l'elemento interessato e non tutta la pagina.
Questa operazione viene effettuata nella RAM confrontando la rappresentazione virtuale precedente con quella successiva alla modifica e quindi risulta molto veloce.

2.4.2 Pregi e difetti

I pregi di questa libreria risiedono nel fatto che offre un insieme completo di strumenti che permettono di gestire l'intero ciclo di vita di un template.
L'arricchimento degli strumenti relativi alla sintassi mustache permette di inserire nel template campi che possono essere risultato di espressioni ed andare a variare sia i dati rappresentati che elementi del CSS, questo permette di creare template che cambiano dinamicamente in base al variare dei dati anche nell'aspetto grafico.
Inoltre la possibilità di creare template che racchiudano HTML, CSS, e comportamento all'interno di un unico file, risulta molto comodo nello sviluppo di quest'ultimi.
Come contro, in relazione alle librerie viste in precedenza si può citare il peso maggiore in termini di risorse e la quantità di tempo neccessaria ad apprendere il funzionamento della libreria.

2.5 Confronto finale

Mettendo a confronto le varie librerie si nota che *Mustache.js* e *HandlebarsJS* sono molto simili sia come supporto alla sintassi mustache che come approccio previsto per la creazione dei template.
Nel confronto fra le due vince sicuramente *HandlebarsJS* per il fatto che offre qualche strumento in più rispetto a *Mustache.js* e la possibilità di poter usufruire degli helpers lo rendono più completo e versatile.
Ractive.js può essere considerata di una categoria superiore rispetto alle due librerie precedenti sia per il suo approccio nella gestione dei template che per il grande numero di funzionalità offerte, che permette la gestione del template in ogni suo aspetto.
Questa libreria può essere utilizzata sia per la creazione di template semplici che molto complessi, inoltre è l'unica delle tre librerie ad offrire la possibilità di rendere interattivi i template tramite il data-binding e la gestione degli eventi.

2.6 Libreria scelta

Non essendo stati stabiliti vincoli sulla creazione dei template, questi possono essere di diversa complessità, possono essere costruiti con linguaggio HTML o SVG e presentare elementi interattivi.
Queste condizioni hanno fatto ricadere la scelta sulla libreria Ractive.js, per le sue caratteristiche che la rendono uno strumento potente e versatile e che non limita le possibilità di sviluppo dei template.

Capitolo 3

Strumenti e tecnologie utilizzati

3.1 Linguaggi utilizzati

I template e l'editor creati durante il periodo di stage sono stati sviluppati per poter essere utilizzati all'interno di un browser, quindi i linguaggi utilizzati sono strettamente legati a questo ambiente.
Questi linguaggi sono i seguenti:

- Il linguaggio HTML è stato utilizzato per creare la struttura dei template e dell'editor;
- Il linguaggio SVG è stato utilizzato nella creazione di template contenenti immagini vettoriali.
- Il linguaggio CSS è stato utilizzato sia per definire la rappresentazione grafica dei template sia dell'editor;
- Il linguaggio JavaScript è stato utilizzato per definire il comportamento dell'editor e per implementare varie funzionalità dei template;
- La sintassi mustache è stata utilizzata all'interno dei template, sia HTML che SVG.

3.2 JavaScript ES5

Il codice JavaScript prodotto durante lo sviluppo del progetto segue gli standard ES5 trattandosi della versione supportata dal JavaScriptCore e compatibile con tutti i browser recenti sia in versione desktop che mobile.
La libreria *Ractive.js* offre il supporto ad ES2015 permettendo di utilizzare le **promise** ed altri elementi che sono stati introdotti nello standard ES6 anche se non supportati dal browser.
Nel caso in cui il browser supporti le **promise**, vengono utilizzate quelle definite nel ES6, altrimenti vengono utilizzate quelle definite dalla libreria che per il momento non supportano *Promise.race* e *Promise.cast*.

3.3 Editors

Per lo svolgimento del progetto poteva essere utilizzato qualsiasi editor di testo ma la scelta è ricaduta su *SublimeText* data la sua versatilità e la sua licenza free.
Questo editor riconosce di default i linguaggi HTML, CSS, e Javascript e offre varie funzionalità come l'auto-completamento del codice, e un set di *bundle* che permettono la stesura del codice in modo automatico tramite l'inserimento di *keyword*.
Inoltre *SublimeText* permette l'estensione delle sue funzionalità tramite l'aggiunta di moduli offerti dalla comunità che lo supporta.

3.4 Inkscape

Inkscape è un software per la grafica vettoriale libero e open, disponibile per varie piattaforme.
Questo software permette la creazione di immagini vettoriali nel formato SVG (Sclable Vector Graphics) e la loro esportazione in più varianti di questo formato.
L'utilizzo di questo strumento è risultato molto utile per la creazione della struttura base di vari template, che in seguito sono stati modificati tramite la sintassi mustache.

3.5 Google Chrome Dev Tools

Sono un insieme di strumenti offerti da Google agli sviluppatori accessibili all'interno del browser Google Chrome.
Questi strumenti risultano molto utili sia nella fase di sviluppo che di testing dell'applicazione perché permettono allo sviluppatore di vedere tutti gli elementi che compongono la pagina, ottenere informazioni sul network, tempi di caricamento delle risorse ed altro.
Inoltre è possibile eseguire il codice passo passo tramite il debuger integrato e utilizzare la console per stampare messaggi (tramite *console.log()*) o eseguire metodi.

3.6 QUnit

QUnit è un framework per il test su script JavaScript.
Normalmente viene utilizzato per effettuare test di unità sulle funzioni o metodi JavaScript a livello di dati ricevuti e restituiti.
Nelle ultime versioni del framework è stata aggiunta la possibilità di effettuare test anche sulla manipolazione del DOM, questa funzione risulta utile per testare script JavaScript che non restituiscono dati ma vanno a manipolare il browser DOM.

Capitolo 4

I template

In questo capitolo viene descritto il lavoro svolto nella prima parte del progetto, che consiste nella realizzazione dei template, nel spiegare come sono stati resi responsive ed in fine come sia stato possibile inserire *plug-in* jQuery al loro interno.
Viene anche trattato l'argomento relativo al loro caricamento all'interno della pagina HTML e la gestione delle librerie.

4.1 Utilizzo di Ractive.js

In questa sezione viene descritta più in dettaglio la libreria scelta per svolgere il progetto.

4.1.1 L'oggetto Ractive

Per iniziare ad usare Ractive bisogna innanzitutto creare un istanza dell'oggetto Ractive e passarle le opzioni desiderate tra quelle offerte dalla libreria.
Una volta creato l'oggetto questo si occuperà di creare e popolare il proprio registro dati e di creare in memoria una rappresentazione virtuale del template.

```
1 var ractive = new Ractive({
2   el: '#container', // id dell'elemento target nella pagina
3   template: '<p>{{greeting}}, {{recipient}}!</p>', // esempio di template
4   data: { greeting: 'Hello', recipient: 'world' } // oggetto JavaScript
      contenente i dati
5 });
```

Codice 4.1: Creazione di un oggetto Ractive.

4.1.2 Le opzioni principali

La libreria fornisce un insieme consistente di opzioni che possono essere inizializzate alla creazione dell'oggetto Ractive.
Queste si dividono in sei categorie che sono le seguenti:

- data;
- templating;
- transitions;
- binding;
- parse;
- lifecycle event.

Tutte queste opzioni opportunamente inizializzate vanno a definire le proprietà ed il comportamento del template nel suo intero ciclo di vita.
Le opzioni principali sono:

- **el**: identifica l'elemento target all'interno del browser DOM dove verrà renderizzato il template;
- **template**: rappresenta il template da renderizzare;
- **data**: rappresenta i dati che dovranno essere interpolati con il template;
- **compute**: oggetto che può contenere espressioni o funzioni che verranno ricalcolate al variare del data model;

4.1.3 Il two-way data biding

La libreria fornisce la funzionalità di one-way-binding tra i dati contenuti nel data registry e il virtual DOM, cioè al variare del model il virtual DOM e di conseguenza anche la sua rappresentazione nel browser vengono modificate.
Oltre a questo tipo di legame, la libreria offre, per elementi di input , il two-way-binding, che permette di modificare il model al variare dei dati nella view e viceversa.

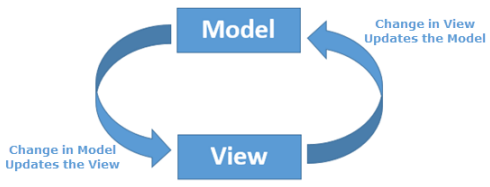


Figura 4.1: Rappresentazione two way data binding.

4.1.4 Gli eventi

La libreria Ractive implementa il pattern architetturale *publish/subscribe*, che permette di rispondere o innescare particolari eventi, che attualmente vengono gestiti su due livelli.
Il primo è un interazione a basso livello con gli eventi del DOM che viene specificata tramite *directives* del template che specificano anche come l'evento deve essere gestito, tramite *proxy event* o chiamate a metodi.
Il secondo è gestito dalle API *publish/subscribe* e dal sistema di eventi all'interno di Ractive e tra i componenti.
I *proxy events* collegano gli eventi del DOM con gli eventi di Ractive, mentre le chiamate a metodi direttamente sull'istanza ractive non utilizzano l'infrastruttura *publish/subscribe*.

4.1.5 Il virtual DOM

Ractive utilizza un sistema differente dagli altri per tracciare le modifiche, ricorrendo al cosiddetto Virtual DOM, ossia a una rappresentazione virtuale della struttura del template immagazzinata in memoria e del tutto simile al DOM originale, del quale può essere vista come una astrazione.
Nel momento in cui si verifica un evento ed è necessario reagire ad esso modificando gli elementi della pagina, Ractive applica prima tali interventi al Virtual DOM.
Attraverso l'analisi delle differenze tra lo stato del Virtual DOM precedente al verificarsi dell'evento e quello nuovo ottenuto dall'applicazione delle modifiche, Ractive determina i cambiamenti effettivi da apportare al DOM vero e proprio.
Il calcolo delle differenze tra i due stati del DOM virtuale è estremamente veloce, e grazie a esso si limitano al minimo indispensabile gli interventi sul DOM reale, tendenzialmente più lento, garantendo quindi ottime performance.

4.1.6 Plug-in di terze parti

I *plug-in* permettono di aumentare le funzionalità offerte dalla libreria Ractive.
Agli sviluppatori è data la possibilità di creare i propri *plug-in* o di scegliere quelli più adatti alle proprie esigenze da una lista presente sul sito di Ractive.js.
Durante lo sviluppo del progetto sono stati utilizzati i *plug-in* ractive-events-tap, per gestire il click/tap sui dispositivi mobili , e ractive-load, per il caricamento tramite protocollo *http* dei template.

4.2 Sviluppo dei template

La creazione dei template, per il progetto, non presentava nessuna restrizione né per la forma né per i contenuti, quindi la decisione di quali template sviluppare era a discrezione del programmatore.
L'unica richiesta, per questa fase del progetto, è stata quella di avere template sia HTML che SVG.

4.2.1 Struttura dei template

I template HTML sono strutturati come una pagina web, cioè sono formati dal codice HTML per quanto riguarda i contenuti, il codice CSS per la loro rappresentazione grafica e JavaScript per definire il loro comportamento.
Il codice HTML rappresentante il template deve essere arricchito tramite la sintassi *mustache* contenente le variabili, le espressioni e tutti gli elementi utili al funzionamento del template.
Inoltre possono essere presenti le direttive necessarie nel caso il template presenti la possibilità di interazione con esso.
Il comportamento del template viene sviluppato tramite gli strumenti offerti dalla libreria Ractive.
La libreria permette di creare un singolo file che contiene al suo interno l'intero template cioè HTML+mustache, CSS e Javascript.
Questo risulta molto vantaggioso per diminuire la quantità di file e migliorare l'organizzazione dei vari template.
Ogni template dovrà inoltre avere un file con estensione *.json* contenente i dati da visualizzare ed eventuali immagini o librerie necessarie per il suo corretto funzionamento.
I vari template sono stati suddivisi in cartelle nel seguente modo:

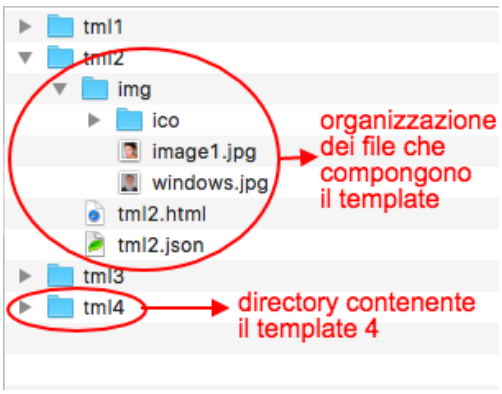


Figura 4.2: Organizzazione dei template nel file system.

4.2.2 Prototipo di template

Di seguito viene riportato una parte del codice utilizzato per la creazione di un template HTML.

```
<div id="tml1" style="background-color: {{bgColor}}; color: {{textColor}};">
  <img class="image" src= {{ img ? img : 'templates/tml1/img/no-image.png'}}>
  <div class="prod-data">
    <h2 class="center">{{nome}}</h2>
    prezzo: <strong>{{prezzoCent}} euro</strong>
    <p class={{ quantita ? 'disponibile' : 'non-disponibile'}}>Disponibilità:
    {{#if quantita}}
      <strong>disponibili {{quantita}}</strong>
    {{else}}
      <strong>Non disponibile</strong>
    {{/if}}
    </p>
    <h3>Descrizione:</h3>
    <p class="desc">{{descrizione}}</p>
  </div>
</div>

<!-- stili -->
<style>
  <!-- parte relativa alle regole CSS -->
  ...
</style>

<script>
  <!-- parte relativa al comportamento del template -->
  component.exports = {
    computed: {
      prezzoCent: function(){
        return this.get('prezzo').toFixed(2);
      }
    }
    ...
  }
</script>
```

Codice 4.2: Esempio di template.

Come si può notare il file è suddiviso in tre sezioni.
La prima contiene il codice HTML con l'aggiunta di variabili ed espressioni inserite tramite mustache.

```
<p class={{ quantita ? 'disponibile' : 'non-disponibile'}} >
```

Codice 4.3: Espressione con mustache.

Nell'esempio sopra riportato il codice permette di modificare l'attributo "classe" del tag *p* in base al valore della variabile *quantita*.
La porzione di codice seguente rappresenta una condizione *if* che visualizza nel template la stringa *"disponibili: numero prodotti"* se *quantita* è maggiore di 0 e *"non disponibile"* altrimenti.

```
  {{#if quantita}}
    <strong>disponibili {{quantita}}</strong>
  {{else}}
    <strong>Non disponibile</strong>
  {{/if}}
```

Codice 4.4: Condizione if-else con mustache.

Il file contiene anche l'implementazione dell'opzione *computed* fornita dalla libreria Ractive.js che verrà utilizzata durante la creazione dell'oggetto Ractive.
Tramite il metodo *export* di component è possibile definire la logica del template utilizzando le opzioni e i metodi forniti dalla libreria Ractive.js.
In questo caso viene indicato che ogni volta che il prezzo subisce modifiche questo deve essere convertito in una stringa e rappresentato con due decimali.

```
component.exports = {
  computed: {
    prezzoCent: function(){
      return this.get('prezzo').toFixed(2);
    }
  }
  ...
}
```

Codice 4.5: Implementazione ed esportazione opzione computed.

Al template viene fornito un oggetto JSON contenente dei dati di default, utilizzati per ottenere una rappresentazione del template completa.
Questi dati sono contenuti in un file con estensione *.json* presente all'interno della directory del template.
L'oggetto JSON è definito come segue.

```
{
  "bgColor":"#ededed",
  "textColor":"#000000",
  "nome": "Cuffie rosse WiFi",
  "img": "templates/tml1/img/cuffie.jpg",
  "prezzo": 200,
  "quantita": 6,
  "descrizione": " ... "
}
```

Codice 4.6: Rappresentazione dell'oggetto JSON.

In seguito vengono proposte due immagini del template descritto in precedenza dopo il rendering da parte della libreria Ractive.js.
Queste immagini fanno vedere come cambia la rappresentazione del template quando la variabile *quantita* è maggiore di 0 e quando è uguale a 0.

Figura 4.3: Rappresentazione del template con *quantità* maggiore di 0.

Figura 4.4: Rappresentazione del template con *quantità* uguale a 0.

Nel caso in cui il template utilizzi SVG, la struttura del template è la stessa, con l'unica differenza che la sintassi mustache viene aggiunta al codice SVG.

4.3 Rendere il template responsive

Uno dei requisiti richiesti per quanto riguarda lo sviluppo dei template è quello di renderli *responsive*, in modo che sia possibile visualizzarli su dispositivi desktop e mobile.
La soluzione individuata per soddisfare questo requisito è stata quella di inserire delle *media-query* all'interno del CSS del template, definendo le regole per la corretta rappresentazione a diverse risoluzioni.
La pagina che andrà ad ospitare i template dovrà contenere il meta-tag *viewport* per garantire la corretta visualizzazione.
Utilizzando questo metodo si riesce a mantenere il CSS all'interno del file che contiene il template, senza ricorrere ad uno o più file esterni contenenti le regole relative a vari dispositivi.

Di seguito viene proposto un esempio di media query.

```
@media only screen and (min-width: 320px) and (max-width: 639px){

  .image{
    width: 90%;
    ...
  }

  .prod-data{
    float: none;
    ..
  }

  ...
}
```

Codice 4.7: Esempio di media query nel CSS del template.

4.4 Inserimento plug-in jQuery nei template

Oltre a rendere i template responsive, un'altra richiesta da soddisfare durante questa fase del progetto consisteva nello studiare la possibilità di sviluppare template che contenessero plug-in jQuery.
L'utilizzo di plug-in jQuery implica:

- il caricamento della libreria del plug-in;
- la stesura del codice HTML seguendo le regole fornite dallo sviluppatore della libreria;
- la stesura di uno script JavaScript che va a richiamare le funzionalità desiderate tra quelle offerte dal plug-in.

Questo particolare script deve essere eseguito in seguito al caricamento del template all'interno della pagina.
La soluzione individuata per avere template che contengano plag-in jQuery consiste nell'inserire lo script per il funzionamento del plug-in come implementazione dell'opzione *oncomplete* dell'oggetto Ractive.
L'opzione *oncomplete* fa parte dei *lifecycle events* di Ractive e viene richiamata nel momento in cui il template è stato completamente renderizzato.
Questa soluzione oltre a permettere l'inserimento dello script all'interno del file del template, assicura che la sua esecuzione avvenga in maniera corretta.
In seguito viene riportato un esempio di implementazione dell'opzione *oncomplete* per l'utilizzo di un plug-in jQuery.

```
<script>
  component.exports = {
    oncomplete: function() {
      jQuery(this.find('#img1')).actuate('wobble');//find metodo di Ractive
      jQuery(this.find('#img2')).actuate('pulse');//actuate metodo plug-in
    }
  }
</script>
```

Codice 4.8: Implementazione dell'opzione *oncomplete*

Ogni template che utilizza plug-in jQuery contiene nella propria directory, oltre alle librerie necessarie anche un file *.json* che contiene un array con le *path* delle librerie da caricare.

4.5 Caricamento dei template nelle pagine HTML

Per effettuare il caricamento dei template all'interno della pagina, sono state sviluppate varie funzioni.
Queste funzioni si distinguono per il tipo di template che devono caricare, cioè:

- caricamento di template senza plug-in jQuery;
- caricamento di template con plug-in jQuery e script di attivazione del plug-in all'interno del file del template;
- caricamento di template con plug-in jQuery e script di attivazione del plug-in come file esterno al file del template;

Quest'ultimo caso è stato inserito per dare una possibilità in più nello sviluppo dei template, anche se non risulta essere la scelta migliore.
Queste funzioni ,una volta ottenuto il nome del template da caricare, si occupano di effettuare una richiesta *http-get* per ottenere i vari componenti del template.
In seguito, utilizzando le risorse ottenute, viene istanziato un oggetto Ractive per ogni template.
Quest'ultimo si occupa di renderizzare il template nella pagina.
Per quanto riguarda il caricamento dei template che utilizzano plug-in jQuery, prima di effettuare la richiesta del template e dei suoi dati, viene effettuato il caricamento delle librerie necessarie al funzionamento del plug-in.
Questo viene fatto leggendo il file *libs.json* relativo al template e inserendo all'interno della pagina la richiesta di caricamento delle librerie.
Su richiesta del tutor aziendale è stata sviluppata una funzione che controlla le librerie che dovranno essere caricate dal browser e nel caso in cui la libreria che deve essere caricata con il template sia già presente non viene effettuata un'ulteriore richiesta di caricamento.
Questo comportamento evita di effettuare richieste superflue nel caso venissero caricati più template che utilizzano la medesima libreria.

In seguito viene proposto il codice delle funzioni che si occupano di gestire il caricamento delle librerie.

```
//funzione che controlla se la libreria fa parte di quelle gia caricate
var confrontaScript = function confrontaScript(library) {
  var scriptArray = document.scripts;
  var trovato = false;
  for (var i = 0; i < scriptArray.length && !trovato; i++) {
    var scriptUrl = scriptArray[i].attributes.src.value;
    var scriptName = scriptUrl.slice( scriptUrl.lastIndexOf('/')+1, scriptUrl
    .length);
    if (scriptName === library) {
      trovato = true;
      console.log('lo script '+library+' è già presente!');
    }
  }
  return trovato;
}

this.confrontaScriptByUrl = function(url) {
  var libraryUrl = url;
  var libraryName = libraryUrl.slice( libraryUrl.lastIndexOf('/')+1,
    libraryUrl.length );
  return confrontaScript(libraryName);
}
// funzione che si occupa se necessario di aggiungere la richiesta di
    caricamento alla pagina
this.addLibraryFromUrl = function(url) {
  var libraryUrl = url;
  var libraryName = libraryUrl.slice( libraryUrl.lastIndexOf('/')+1,
    libraryUrl.length );
  if (!confrontaScript(libraryName)) {
    var node = document.createElement('script');
    node.setAttribute('src', libraryUrl);
    document.head.appendChild(node);
    console.log('libreria '+libraryName+' aggiunta al file HTML');
  }
  else {
    console.log('libreria '+libraryName+' gia presente nel file HTML');
  }

}
```

Codice 4.9: Funzioni che si occupano del caricamento delle librerie

4.5.1 Problemi riscontrati

Un problema che è stato rilevato, a riguardo del caricamento di template che utilizzano jQuery consiste nel fatto che il browser effettua il caricamento delle librerie in modo asincrono, quindi non gestibile da parte dello sviluppatore.
Nel caso in cui la libreria non sia ancora stata caricata prima dell'avvenuto rendering del template, si verifica un errore perché le funzioni richiamate non vengono riconosciute.
Il problema riscontrato sarebbe risolvibile se fosse possibile rilevare l'avvenuto caricamento delle risorse richieste, ma momentaneamente i browser non offrono strumenti adeguati.
Una delle possibili soluzioni è quella di effettuare il *reload* della pagina una volta che questa è stata completamente caricata.

Capitolo 5

Analisi dei Requisiti

In questo capitolo sono contenuti i requisiti dell'applicazione che sono stati individuati durante il progetto.
Le linee guida per la creazione dell'applicazione sono state fornite dal tutor e sulla base di esse sono stati individuati i requisiti che in seguito sono stati dicussi con il tutor per ottenerne l'approvazione.

5.1 Applicazione per la modifica dei template

L'applicazione richiesta per il progetto deve permettere all'utente di visualizzare un insieme di template predefinito, da cui sia possibile selezionare quello desiderato.
In seguito alla selezione del template, l'utilizzatore deve poter visualizzare quest'ultimo all'interno di una *view* apposita.
In questa fase deve essere fornito all'utente un editor specifico in relazione al template selezionato, che offra la possibilità di visualizzare e modificare i dati forniti di *default* dal template e di vedere all'interno della *view* dedicata il comportamento del template in seguito alla modifica dei dati.
L'applicazione deve eseguire all'interno di un browser e deve essere compatibile con i più importanti fra essi (Chrome, Firefox, Opera ed Edge).

5.1.1 Visualizzazione lista dei template

Questa sezione dell'applicazione è dedicata alla visualizzazione e selezione dei template disponibili.
La lista deve essere composta dalle miniature dei template in modo da offrire una prima visione di come viene rappresentato il template.
All'utente deve essere permesso di scorrere tutta la lista tramite uno scroll infinito.
La lista deve presentare i template per categorie, le seguenti sono quelle individuate durante l'analisi:

- template singoli;
- template composti;
- template singoli contenenti plug-in JQuery;
- template composti contenenti plug-in JQuery;

I template *singoli* sono dei template la cui costruzione è frutto dell'interpolazione dei dati con un singolo template, mentre quelli *composti* sono template frutto dell'unione di più template.

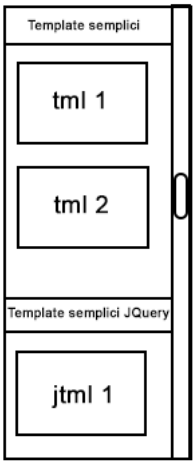


Figura 5.1: Mockup lista di visualizzazione template.

5.1.2 Visualizzazione template selezionato

Questa sezione dell'applicazione è dedicata alla visualizzazione del template selezionato. Il box di visualizzazione non presenta particolari caratteristiche, la sua funzione è quella di mostrare all'utente il template selezionato come verrebbe visualizzato nella pagina html e permettere di osservare i cambiamenti del template in base alla modifica dei dati.

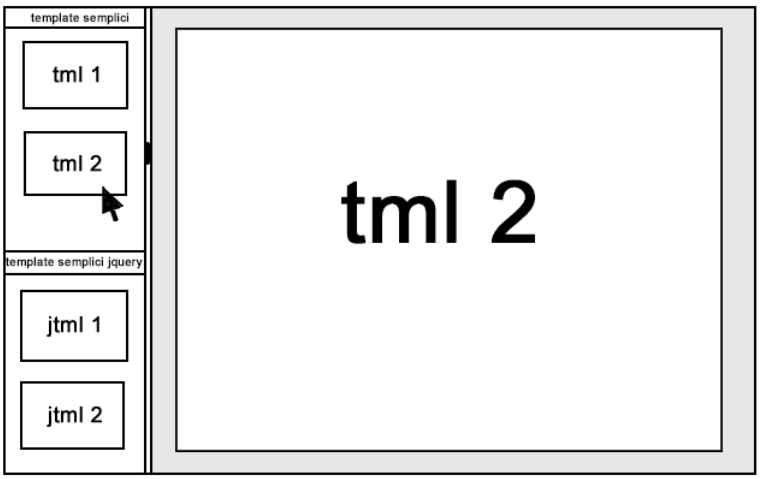


Figura 5.2: Mockup box di visualizzazione template.

5.1.3 Editor per la modifica del template

Questa è l'ultima sezione dell'applicazione e si occupa di fornire all'utente gli strumenti per la modifica dei dati relativi al template selezionato.
L'utente deve poter visualizzare tutti i dati disponibili per la modifica tramite un interfaccia visuale.
Quest'ultima deve fornire all'utente gli strumenti più adeguati per la modifica dei dati in base al loro tipo.
Nel caso in cui i template siano composti, l'editor deve visualizzare l'oggetto JSON che rappresenta i dati e dare all'utente la possibilità di modificarne il codice.

Fig. 5.3: Editor template semplici.

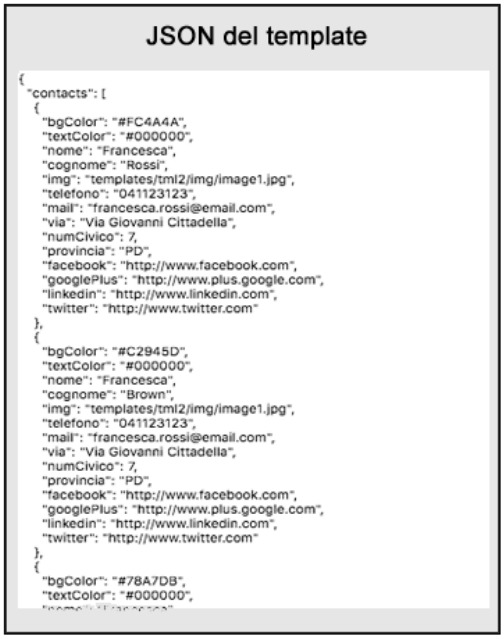


Fig. 5.4: Editor template composti.

5.2 Requisiti individuati

I requisiti individuati sono frutto dall'analisi ed espansione dei requisiti di base e delle discussioni con il tutor aziendale.
Di seguito viene descritto il codice con cui sono stati catalogati:

R[T][I][C]

dove:

- **T**ipo: specifica la tipologia del requisito e può assumere i seguenti valori:
  - **F** - *funzionale*, cioè che determina una funzionalità dell'applicazione;
  - **V** - *vincolo*, che riguarda un vincolo che il prodotto deve rispettare.
- **I**mportanza: specifica l'importanza del requisito e può assumere i seguenti valori:
  - **O** - *obbligatorio*, il requisito corrisponde ad un obbiettivo minimo del piano di stage e deve essere soddisfatto per garantire il funzionamento minimo dell'applicazione;
  - **D** - *desiderabile*, il requisito corrisponde ad un obbiettivo massimo del piano di stage e deve essere soddisfatto per garantire il funzionamento dell'applicazione;
  - **F** - *facoltativo*, indica che il requisito fornisce del valore aggiunto all'applicazione e non era stato previsto nel piano di stage.
- **C**odice: rappresenta un codice che identifica il requisito all'interno di una gerarchia. Questo codice è definito in modo che il requisito *RTIx.y* sia un requisito che va a definire con un grado maggiore di dettaglio alcuni degli aspetti del requisito *RTIx*.

5.2.1 Requisiti Funzionali

Id Requisito	Descrizione
RFO1	L'utente deve poter visualizzare la lista dei template offerti dall'applicazione
RFO1.1	L'utente deve poter visualizzare le miniature dei template
RFF1.1.1	La comparsa delle miniature all'interno della lista deve avvenire in modo animato
RFO1.2	L'utente deve poter scorrere la lista dei template
RFO1.3	L'utente deve poter visualizzare la categoria dei template
RFO1.4	L'utente deve poter selezionare i template
RFD1.4.1	La selezione del template deve avvenire in modo animato
RFO2	L'utente deve visualizzare il template selezionato in un apposito view-box
RFF2.1	La comparsa del template nel view-box deve avvenire in modo animato
RFD2.2	Se il template contiene plug-in JQuery l'utente deve poter visualizzare l'esecuzione del plug-in
RFO2.3	L'utente deve visualizzare nel view-box l'effetto delle modifiche appotrate al template
RFO2.3.1	L'utente deve poter visualizzare l'effetto sul template delle modifiche ai dati di tipo colore

Id Requisito	Descrizione
RFO2.3.2	L'utente deve poter visualizzare l'effetto sul template delle modifiche ai dati di tipo numero
RFO2.3.3	L'utente deve poter visualizzare l'effetto sul template delle modifiche ai dati di tipo booleano
RFO2.3.4	L'utente deve poter visualizzare l'effetto sul template delle modifiche ai dati di tipo immagine
RFO2.3.5	L'utente deve poter visualizzare l'effetto sul template delle modifiche ai dati di tipo url
RFO2.3.6	L'utente deve poter visualizzare l'effetto sul template delle modifiche ai dati di tipo mail
RFO2.3.7	L'utente deve poter visualizzare l'effetto sul template delle modifiche ai dati di tipo stringa breve
RFO2.3.8	L'utente deve poter visualizzare l'effetto sul template delle modifiche ai dati di tipo testo
RFO2.3.9	L'utente deve poter visualizzare l'effetto sul template delle modifiche ai dati di tipo JSON
RFO3	L'utente deve poter visualizzare i dati di default forniti dal template selezionato
RFO3.1	L'utente deve poter visualizzare i dati di tipo colore
RFO3.2	L'utente deve poter visualizzare i dati di tipo numero
RFO3.3	L'utente deve poter visualizzare i dati di tipo booleano
RFO3.4	L'utente deve poter visualizzare i dati di tipo immagine
RFO3.5	L'utente deve poter visualizzare i dati di tipo url
RFO3.6	L'utente deve poter visualizzare i dati di tipo mail
RFO3.7	L'utente deve poter visualizzare i dati di tipo stringa breve
RFO3.8	L'utente deve poter visualizzare i dati di tipo testo
RFO3.9	L'utente deve poter visualizzare i dati in formato JSON
RFO4	L'utente deve poter modificare i dati di default forniti dal template selezionato
RFO4.1	L'utente deve poter modificare i dati di tipo colore
RFD4.1.1	L'utente deve poter visualizzare il color-picker
RFF4.1.1.1	La comparsa del color-picker deve avvenire in modo animato
RFF4.1.1.2	La scomparsa del color-picker deve avvenire in modo animato
RFD4.1.2	L'utente deve poter selezionare il colore nel color-picker
RFO4.2	L'utente deve poter modificare i dati di tipo numero
RFO4.3	L'utente deve poter modificare i dati di tipo booleano
RFO4.4	L'utente deve poter modificare i dati di tipo immagine
RFD4.4.1	L'utente deve poter caricare un'immagine da filesystem
RFO4.5	L'utente deve poter modificare i dati di tipo url
RFO4.6	L'utente deve poter modificare i dati di tipo mail
RFO4.7	L'utente deve poter modificare i dati di tipo stringa breve
RFO4.8	L'utente deve poter modificare i dati di tipo testo
RFO4.9	L'utente deve poter modificare i dati dell'oggetto in formato JSON
RFD5	L'utente deve poter visualizzare un messaggio di errore nel caso in cui l'inserimento dei dati avvenga in maniera non corretta

Tabella 5.1: Requisiti Funzionali

5.2.2 Requisiti di Vincolo

Id Requisito	Descrizione
RVO1	L'applicazione deve utilizzare HTML5
RVO2	L'applicazione deve utilizzare CSS3
RVO3	L'applicazione deve utilizzare JavaScript
RVO4	L'applicazione deve utilizzare Ractive.js
RVO5	L'applicazione deve funzionare su *Google Chrome* versione 52.0 o superiore
RVO6	L'applicazione deve funzionare su *Firefox* versione 46.0 o superiore
RVD7	L'applicazione deve funzionare su *Safari* versione 9.0 o superiore
RVD8	L'applicazione deve funzionare su *Edge* versione 37.0 o superiore
RVD9	L'applicazione deve funzionare su *Opera* versione 37.0 o superiore
RVF10	L'applicazione deve funzionare su *Internet Explorer* versione 11.0 o superiore

Tabella 5.2: Requisiti di Vincolo

5.3 Riepilogo requisiti

I requisiti individuati sono in totale 56 e vengono ripartiti tra le varie tipologie secondo quanto riportato nelle seguenti tabelle.

Importanza	#
Obbligatori	42
Desiderabili	9
Facoltativi	5
Totale	56

Tabella 5.3: Numero di requisiti per importanza

Tipologia	#
Funzionali	46
Vincolo	10
Totale	56

Tabella 5.4: Numero di requisiti per tipologia

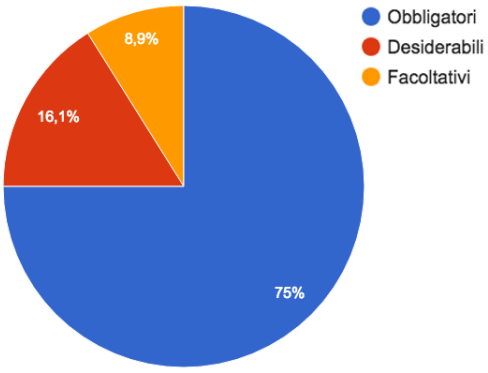


Figura 5.5: Requisiti per importanza

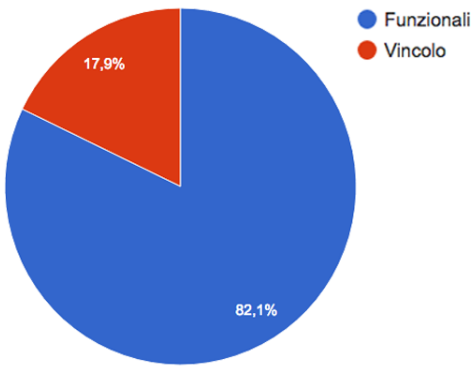


Figura 5.6: Requisiti per tipologia

Capitolo 6

Progettazione

In questo capitolo viene spiegato il metodo di suddivisione delle risorse, come viene effettuato il caricamento dei vari componenti dei template e vengono descritte le funzioni che compongono l'applicazione.

6.1 Suddivisione template

Come descritto nel capitolo *Analisi dei requisiti* 5.1.1 i template sono stati divisi in quattro tipologie.
Dato che non sono presenti delle API fornite dall'azienda, per ottenere informazioni sui template disponibili, come il loro numero, il loro tipo e le risorse che li compongono, è stato deciso di utilizzare un metodo semplice per gestire i template all'interno del file-system.
Le risorse sono state suddivise in directory, che rappresentano i template.
Ad ogni directory e alle risorse principali in esse contenute è stato assegnato un nome che permette di identificare il singolo template e il suo tipo.
Le quattro categorie vengono nominate come segue:

- **tml** identifica i template di tipo semplici;
- **jtml** identifica i template di tipo semplici contenenti plug-in JQuery;
- **ctml** identifica i template di tipo composti;
- **jctml** identifica i template di tipo composti contenenti plug-in JQuery.

6.1.1 Accesso alle risorse

Le risorse necessarie sono reperibili tramite chiamata *http-get* ad un URL nella forma `/templates/{tipo+indice}/{tipo+indice}.estensione`, dove `tipo` è uno dei quattro tipi descritti in precedenza, `indice` è il numero che identifica il template.

GET /templates/{tipo+indice}/{tipo+indice}.html

Restituisce il file HTML che descrive il template, nel caso in cui l'operazione non vada a buon fine viene lanciato un errore di tipo `404` e un mesaggio *file non trovato*.

```
<div id="tml1" style="background-color: {{bgColor}}; color: {{textColor}};">

  <img class="image" src= {{ img ? img : 'templates/tml1/img/no-image.png'}}>
  <div class="prod-data">
    <h2 class="center">{{nome}}</h2>
    prezzo: <strong>{{prezzoCent}} Euro</strong>
    <p class={{ quantita ? 'disponibile' : 'non-disponibile'}} >Disponibilità
    : {{#if quantita}}
      <strong>disponibili {{quantita}}</strong>
      {{else}}
      <strong>Non disponibile</strong>
      {{/if}}
    </p>
    <h3>Descrizione:</h3>
    <p class="desc">{{descrizione}}</p>
  </div>
</div>

<!-- stili -->
<style>
#tml1{
  position: relative;
  font-family: Arial, Verdana, Helvetica, sans-serif;
  border: 1px solid black;
  overflow: auto;
}

...

</style>

<!-- comportamento -->
<script>
  component.exports = {
    computed: {
      prezzoCent: function(){
        return this.get('prezzo').toFixed(2);
      }
    }

    ...
  }
</script>
```

Codice 6.1: Esempio di template mustache restituito.

GET /templates/{tipo+indice}/{tipo+indice}.json

Restituisce l'oggetto JSON contenente i dati del template, nel caso in cui l'operazione non vada a buon fine viene lanciato un errore di tipo `404` e un mesaggio *file non trovato*.

```
{
  "bgColor":string,
  "textColor":string,
  "nome": string,
  "img": string,
  "prezzo": number,
  "quantita": number,
}
```

Codice 6.2: Esempio di oggetto JSON restituito.

GET /templates/{tipo+indice}/{tipo+indice}_libs.json

Restituisce, se il template contiene plug-in, l'oggetto JSON contenente l'URL delle librerie utilizzate, nel caso in cui l'operazione non vada a buon fine viene lanciato un errore di tipo `404` e un mesaggio *file non trovato*.

```
{
  "libs": [
    string,
    string,
    ...
  ]
}
```

Codice 6.3: Esempio di oggetto JSON restituito.

6.2 Caricamento template

Il caricamento di un template consiste nel reperire le risorse principali ad esso relative e con queste istanziare un oggetto di tipo ractive, che una volta creato verrà renderizzato in un apposito elemento HTML.
Per effettuare questa operazione sono state definite le seguenti funzioni:

- `loadTemplateWithoutJQuery(tmlUrl, dataUrl, tmlAnchor)`
  Questa funzione si occupa del caricamento e del rendering dei template che *non contengono* plug-in JQuery.
  La funzione recupera le risorse da `tmlUrl` e `dataUrl`, crea un'istanza *Ractive* che rappresenta il template e lo renderizza all'interno dell'elemento HTML con id `tmlAnchor`.

- `loadTemplateWithJQueryPlugins(tmlUrl, dataUrl, libsUrl, tmlAnchor)`
  Questa funzione si occupa del caricamento e del rendering dei template che *contengono* plug-in JQuery.
  La funzione recupera la lista delle librerie JQuery del template da `libsUrl` e usa la funzione `loadLibs` per gestire il loro caricamento.
  In seguito recupera le risorse da `tmlUrl` e `dataUrl`, crea un'istanza *Ractive* che rappresenta il template e lo renderizza all'interno dell'elemento HTML con id `tmlAnchor`.

- `loadLibs(libs)`
  Questa funzione riceve un *array* di URL relative alle librerie JQuery da caricare e utilizzando la funzione `addLibraryFromUrl(libUrl)` ne gestisce il caricamento.

```
{
  "libs": [
    "templates/template_name/lib/library1_name.js",
    "templates/template_name/lib/library2_name.js",
    "templates/template_name/lib/library3_name.js"
  ]
}
```

Codice 6.4: Esempio array `libs`.

- `addLibraryFromUrl(libUrl)`
  Questa funzione provvede al caricamento della libreria presente all'URL `libUrl` nel caso questa non sia stata caricata.

6.3 Creazione lista template

In questa sezione vengono descritte le funzioni che si occupano della creazione della lista di selezione dei template.
Questa lista deve essere composta dalle miniature dei template suddivise fra le quattro categorie descritte in precedenza.
Per poter effettuare il caricamento è necessario definire un oggetto JavaScript che contenga il numero di template disponibili suddivisi per categoria.
La scelta di definire questo oggetto è stata dettata dalla necessità di conoscere l'insieme dei template disponibili.
Data la struttura del file-system e il fatto che i template sono indicizzati in maniera ordinata è possibile descrivere l'insieme dei template da caricare tramite un oggetto JavaScript.

- `templatesToLoad`
  Questo oggetto descrive l'insieme dei template offerti dall'applicazione.
  La sua struttura è formata da coppie del tipo *chiave-valore* che rappresentano rispettivamente:
  - **chiave** definisce il tipo dei template;
  - **valore** contiene il numero dei template di quel tipo.

  L'oggetto è rappresentato come segue:

```
{
  tml: number,
  jtml: number,
  ctml: number,
  jctml: number
}
```

Codice 6.5: Struttura oggetto `templatesToLoad`.

- `createTemplateList()`
  Questa funzione si occupa di inserire le varie categorie all'interno della lista e tramite l'ausilio della funzione `loadTemplateList(type, num)` di popolare la lista.
  La funzione utilizza l'oggetto `templateToLoad` per conoscere l'insieme dei template.

- `loadTemplateList(type, num)`
  Questa funzione si occupa di creare la miniatura e inserirla nella lista in base ai parametri `type` e `num` che compongono il nome del template.

6.4 Visualizzazione template selezionato

In questa sezione viene descritto il comportamento previsto e le funzioni individuate per la gestione del *view-box* che andrà a visualizzare il template selezionato.
Alla base del *view-box* c'è un elemento HTML di tipo `div` con id `"tml-anchor"` che farà da contenitore per i template.
Ogni miniatura presente nella lista in seguito all'evento `click()` invoca una funzione che si occupa di visualizzare il template nel *view-box*.

- `selectTml(el)`
  Questa funzione riceve il nome del template da caricare dal parametro `el` ed in seguito si occupa di effettuare il caricamento delle risorse e il rendering del template nel `div` prestabilito.

6.5 Editor per la modifica del template

In seguito alla selezione di un template questo deve essere visualizzato e sulla base dei suoi dati modificabili deve essere costruito un editor che ne permetta la modifica.
In questa sezione vengono descritte le funzioni che si occupano di creare l'editor per la modifica dei dati del template.
La creazione dell'editor nelle sue diverse versioni (per template semplici e composti) viene gestita dalle seguenti funzioni:

- `creaTml(type,dati)`
  Questa funzione riceve il tipo del template selezionato tramite il parametro `type` e l'oggetto JSON contenente i dati del tramite il parametro `dati`.
  La funzione costruisce l'editor in base al tipo del template, utilizzando le funzioni `creaEditorFullJson( dati, editorAnchor )` e `creaEditor( dati, editorAnchor )` che verranno descritte in seguito.

- `creaEditorFullJson(obj, el)`
  Questa funzione è rivolta ai template di tipo *composto* e si occupa di creare un area editabile contenete il codice che descrive l'oggetto JSON fornito con il parametro `obj` e ne permette la modifica.
  Gli elementi creati dalla funzione vengono inseriti nell'elemento HTML con id `el`.

- `creaEditor(obj, el)`
  Questa funzione è rivolta ai template di tipo *semplice* e si occupa di creare l'editor contenente tutti gli strumenti per la modifica di ogni dato all'interno del parametro `obj`.
  L'editor viene inserito all'interno dell'elemento HTML con id `el` indicando il nome del dato seguito dallo strumento per la sua modifica.
  Durante la creazione dell'editor viene utilizza la funzione `parse` per la costruzione dei vari strumenti che andranno a comporre l'editor.

- `parse( obj, el, index, regExpArray, path )`
  Questa funzione riceve l'oggetto JSON tramite il parametro `obj` e utilizzando i parametri `index`, `regExpArray` e `path` come controllo, crea lo strumento più adeguato per ogni tipo di dato individuato aggiungendo ognuno di essi all'elemento HTML con id `el`.
  Gli strumenti creati dalla funzione offrono la possibilità di modifica per i dati di tipo:

  - **colore** tramite color picker;
  - **numero** tramite input number;
  - **booleano** tramite combobox (true,false);
  - **immagine** tramite selettore di immagini da file-system;
  - **url** tramite line-edit;
  - **mail** tramite line-edit;
  - **sringa breve** tramite line-edit;
  - **testo** tramite text-area;

Capitolo 7

Realizzazione

In questo capitolo vengono descritte le attività svolte durante lo sviluppo dell'applicazione e le principali difficoltà riscontrate.
Per lo sviluppo dell'applicazione sono state utilizzate, riadattandole, anche soluzioni sviluppate durante la fase di studio sui template, in particolare quelle relative al caricamento dei template e alla gestione delle librerie JQuery.
Durante questa fase il lavoro svolto è stato proposto al tutor aziendale in più riprese perché ne verificasse il comportamento e proponesse eventuali modifiche.

7.1 Il caricamento dei template

Il caricamento dei template consiste in tre fasi, che sono:

- caricamento delle risorse;
- creazione istanza *Ractive*;
- rendering del template all'interno di un elemento HTML.

Per effettuare il caricamento delle risorse sono stati utilizzati due metodi differenti in base al tipo di file da caricare.
Il caricamento degli oggetti di tipo JSON, come i dati e l'elenco delle librerie JQuery del template, è stato effettuato tramite un metodo offerto dalla libreria JQuery, che permette di effettuare una *GET HTTP request* per il caricamento specifico di oggetti JSON.
Il metodo in questione è `jQuery.getJSON()` che effettua una *callback* ad un URL e ritorna l'oggetto desiderato.
Per effettuare il caricamento del template mustache invece, la comunità di Ractive offre un plug-in chiamato `ractive-load`[1] che aggiunge un metodo statico alla libreria e permette, tramite la *promise*[2] `Ractive.load()` di effettuare il caricamento del file contenente la definizione del template utilizzando *GET HTTP request*.

[1] https://github.com/ractivejs/ractive-load
[2] http://www.ecma-international.org/ecma-262/6.0/#sec-promise-objects

```
// carico i dati del template
$.getJSON(dataUrl, function(dati) { // se il caricamento ha successo
  // carico il template tramite Ractive.load
  Ractive.load(tmlUrl).then( function(Template) {
    // creo l'oggetto ractive
    var ractive = new Template({
      el: tmlAnchor,
      data: dati
    });
  ...

  });
})
.fail( function() { // errore caricamento, file non valido
  console.log('file non trovato o errore di caricamento!');
});
```

Codice 7.1: Chiamate *GET HTTP* per il caricamento delle risorse.

Le richieste vengono eseguite in modo asincrono, quindi solamente l'esito positivo della prima *callback* permette l'esecuzione della chiamata a `Ractive.load()` e l'eventuale istanziazione dell'oggetto *Ractive*.
Per quanto riguarda il caricamento di template contenenti plug-in JQuery, il metodo è identico, ma prima di caricare i dati ed il template devono essere caricate le librerie.
Il caricamento delle librerie viene effettuato tramite `JQuery.getJSON()` del file contenente la lista delle librerie e aggiungendo le URL di quest'ultime all'`header` dell'applicazione tramite la creazione di un tag `script` per ogni libreria individuata.

```
// carico le librerie del template
$.getJSON(libsUrl, function(libs) { // se il caricamento ha successo
  // aggiungo le librerie alla pagina HTML
  scriptControll.loadLibs(libs);

  // carico i dati del template
  $.getJSON(dataUrl, function(dati) { // se il caricamento ha successo
    ...
    // istanziazione oggetto ractive
})
.fail( function() { // librerie non trovate o errore di caricamento
  console.log('file non trovato o errore di caricamento!');
////////////////////////////////////////////////////////////////////////
// esempio oggetto JSON contenente le URL delle librerie
{
  "libs": [
    "templates/jtml1/lib/actuate-animate.min.js",
    "templates/jtml1/lib/jquery.drawsvg.min.js"
  ]
}
////////////////////////////////////////////////////////////////////////
// esempio di risultato prodotto nell'HTML
<head>
    ...
  <!-- script aggiunti -->
  <script src="templates/jtml1/lib/actuate-animate.min.js"></script>
  <script src="templates/jtml1/lib/jquery.drawsvg.min.js"></script>
</head>
```

Codice 7.2: Codice per l'aggiunta delle librerie più esempio di JSON e risultato ottenuto.

Per i template con plug-in JQuery il problema principale è quello che il template venga renderizzato prima del caricamento delle librerie, questo comporta il non riconoscimento delle funzioni che si riferiscono al plug-in rendendo il template incompleto.
Quindi il caricamento delle librerie viene eseguito sempre prima di caricare gli altri elementi del template.
Nonostante questa accortezza risulta impossibile verificare l'effettivo caricamento da parte del *browser* delle librerie perché esso viene effettuato in modo asincrono.
Questo problema può essere risolto in maniera semplice con un *reload* della pagina o come è stato fatto in un fork dell'applicazione tramite un *preload* di tutte le librerie, che però risulta una soluzione molto onerosa e in certi casi non risolve il problema.

7.1.1 Controllo delle librerie caricate

Uno dei problemi che è sorto durante lo sviluppo in relazione al caricamento delle librerie per i template con plug-in JQuery, è quello di effettuare il caricamento di una o più librerie già caricate in precedenza.
Questo comporta la manipolazione inutile del *DOM*, quindi una quantità maggiore di carico per il *browser*.
Il problema è stato risolto effettuando un controllo tramite l'implementazione della funzione `confrontaScript()` che utilizza il metodo `document.scripts` per ricavare la lista delle librerie caricate dall'applicazione e tramite un confronto con essa decide se sia necessario aggiungere la nuova libreria o meno.
Questa funzione viene invocata dalla funzione `addLibraryFromUrl()` per ogni URL presente nel JSON, ogni qual volta venga caricato un template contenente plug-in.

```
1  confrontaScript(library) {
2    var scriptArray = document.scripts; // array degli script caricati
3    var trovato = false;
4    for (var i = 0; i < scriptArray.length && !trovato; i++) {
5      var scriptUrl = scriptArray[i].attributes.src.value;
6      var scriptName = scriptUrl.slice( scriptUrl.lastIndexOf('/')+1, scriptUrl
       .length);
7      if (scriptName === library) {
8        trovato = true;
9        //console.log('lo script '+library+' è già presente!');
10     }
11   }
12   return trovato;
13 }
```

Codice 7.3: Funzione che gestisce il caricamento di una libreria.

7.2 Visualizzatore lista template

Dopo aver sviluppato le funzioni per il caricamento dei template è iniziata la fase di realizzazione della sezione adibita alla visualizzazione della lista dei template disponibili.
Il problema che si è presentato prima di iniziare lo sviluppo è stato quello di scegliere quali elementi utilizzare per creare la lista.
La scelta più adeguata in questo caso sarebbe stata quella di caricare una lista di thumbnail (miniature) rappresentanti i vari template disponibili.
Questa decisione però, pur essendo la più efficiente, è stata scartata perché avrebbe

richiesto la creazione di tutte le miniature e l'aggiornamento delle directory dei template.
Per questioni di tempo è stato deciso di caricare direttamente i template nella lista, visto che l'efficienza non era uno dei requisiti importanti per il tutor.
Dopo aver discusso il problema e definito il modo di procedere è iniziato lo sviluppo di questa sezione dell'applicazione.
La costruzione della lista viene effettuata creando i vari elementi che la compongono in modo sequenziale e inserendoli all'interno elemento HTML predisposto a contenerli.
Questa operazione viene effettuata dalle funzioni `createTemplateList()`, che tramite l'oggetto `templatesToLoad` si occupa di suddividere la lista fra le varie tipologie di template, e `loadTemplateList()`, che crea l'elemento `<li>` dedicato a contenere il template.
Il rendering viene effettuato all'interno dell'elemento `<li>` creato in precedenza utilizzando il suo id (creato in maniera univoca) come parametro di ancoraggio per l'oggetto *Ractive* rappresentante il template.

```
function loadTemplateList(type, num) {
  // per ogni tipo di template li carico tutti
  for (var i = 1; i < num+1; i++) {
    // creo list item ancora
    var listItem = "<li class='tml-list-item' ><div class='tml-list-item-div'
    ><a class='tml-list-item-a' onclick='selectTml(this)' href='javascript:
    void(0)' id='"+type+i+"'></a></div></li>";
    // appendo l'elemento alla lista
    $('#tml-list').append(listItem);

    var anchor = '#'+type+i;
    // creo l'oggetto ractive col template relativo
    if (type === 'tml' || type === 'ctml') { // template senza jQuery
      var html = 'templates/'+type+i+'/'+type+i+'.html';
      var dati = 'templates/'+type+i+'/'+type+i+'.json';
      // carico il template
      tl.loadTemplateWithoutJQuery(html, dati, anchor);
    }
    else if (type === 'jtml' || type === 'jctml') { // template senza jQuery
      var html = 'templates/'+type+i+'/'+type+i+'.html';
      var dati = 'templates/'+type+i+'/'+type+i+'.json';
      var libs = 'templates/'+type+i+'/'+type+i+'_libs.json';
      // carico il template
      tl.loadTemplateWithJQueryPlugins(html, dati, libs, anchor);
    }

  }
}
```

Codice 7.4: Implementazione `loadTemplateList()`.

7.3 Visualizzatore template selezionato

In questa fase di sviluppo è stata realizzata la *view* dedicata alla visualizzazione del template selezionato.
Per effettua la visualizzazione è stato necessario:

- individuare il template selezionato all'interno della lista;
- create un'istanza *Ractive* che lo rappresenti;

- renderizzare il template all'interno della *view*.

Per individuare il template selezionato è stato implementato l'evento `onclick` dei vari elementi appartenenti alla lista in modo che richiamino la funzione `selectTml()` passandogli il riferimento all'elemento che ha lanciato l'evento.
Dal riferimento è possibile risalire al nome del template da visualizzare e quindi creare un oggetto *Ractive* che verrà renderizzato all'interno della *view*.
Nel momento in cui viene visualizzato il template, viene richiamata la funzione che si occupa della costruzione del relativo editor, il cui sviluppo verrà descritto nella sezione seguente.

```
<li class='tml-list-item' >
  <div class='tml-list-item-div'>
    <a class='tml-list-item-a' onclick='selectTml(this)' href='javascript:
    void(0)' id='"+type+i+"'></a>
  </div>
</li>
```

Codice 7.5: Chiamata di `selectTml()` all'evento `onclick` del list item.

Le modifiche ai dati sono visualizzate in tempo reale poiché la libreria *Ractive.js* si occupa di effettuare l'update del render in maniera automatica.

7.4 Editor per la manipolazione del template

La realizzazione dell'editor per la modifica dei dati conclude la fase di sviluppo.
L'applicazione deve fornire un editor adeguato ad ogni possibile template, quindi risulta necessario individuare i dati modificabili del template e il loro tipo per poter fornire gli strumenti adeguati per la loro modifica.
Durante questa fase sono state individuate varie soluzioni per risolvere questo problema.
Una di queste consisteva nell'effettuare il parsing del *DOM* del template, individuando il tipo del dato da modificare in base al *tag* HTML che lo conteneva.
Questa soluzione è stata scartata perché troppo onerosa da implementare e perché certi dati all'interno del template potevano essere frutto di espressioni o di funzioni implementate nella logica del template, fatto non deducibile da un parsing del *DOM*.
La soluzione individuata consiste nell'effettuare il parsing dell'oggetto JSON contenente i dati del template, questo offre la sicurezza sulla quantità di dati modificabili, ma fa sorgere un'altro problema.
Il problema riscontrato consiste nel non poter rilevare vari tipi tra quelli descritti nella sezione *Analisi dei requisiti*, come, per esempio i tipi `colore`, `immagine`, `URL`, perché non presenti fra i tipi primitivi in un oggetto JSON.

7.4.1 Creazione dell'editor

La realizzazione dell'editor viene effettuata dalla funzione `creaTml()` che, in base al tipo di template, decide se creare un editor per template semplici o composti.
Per template composti è stata implementata la funzione `creaEditorFullJson()` che si occupa di creare una *text-area* contenente l'oggetto JSON e ne permette la modifica direttamente da codice.
Lo sviluppo dell'editor per i template semplici, che viene effettuato dalla funzione `creaEditor()`, è stato più complesso perché ha richiesto l'individuazione del tipo di

ogni dato presente nel JSON e la creazione del relativo strumento.
Per effettuare la creazione dei vari strumenti e per individuare i tipi di dato è stata sviluppata la funzione `parse()` che, in modo ricorsivo, effettua il parse dell'oggetto JSON individuando il tipo dei dati e costruendo per ognuno di essi il relativo strumento di input.
Il problema della rilevazione dei tipi non primitivi è stato risolto tramite il confronto dei dati di tipo `string` con espressioni regolari che rappresentano i seguenti tipi:

- **colore**, rappresentato sia in formato *rgb* che esadecimale;
- **URL**, rappresentato come url generica a risorse web non di tipo immagine;
- **mail**, rappresentato come indirizzo di posta elettronica;
- **immagine**, rappresentato come url a risorse di tipo immagine nei formati *jpg, jpeg, png, gif* e *svg*.

Nel caso in cui il tipo rilevato sia un immagine viene costruito un input di tipo *input-file* che permette il caricamento di un immagine da file-system.
Mentre per il tipo **colore** è stato utilizzato il plug-in JQuery jscolor, che permette di visualizzare il colore e propone un utile *color-picker* per effettuarne la modifica.

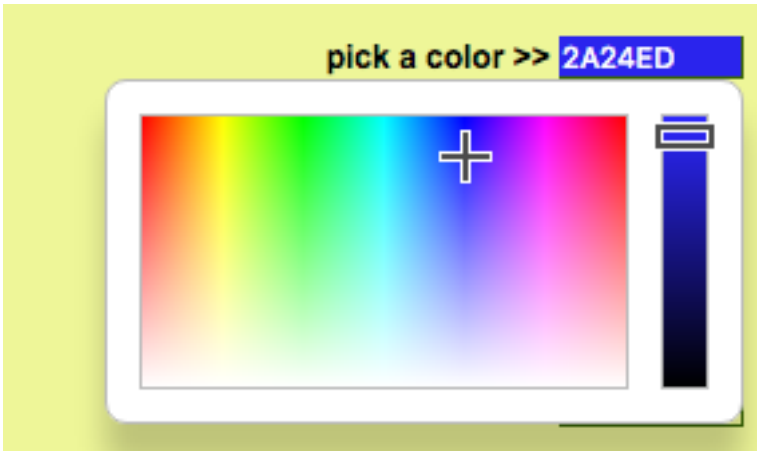


Figura 7.1: Color-picker per la modifica dei tipi colore.

Per la modifica degli altri tipi vengono utilizzati gli strumenti di input forniti dal linguaggio HTML senza ricorrere a librerie esterne.
In seguito viene proposto un estratto della funzione `parse()`.

```
function parse(obj, el, index, regExpArray, path ){

  if (obj == null || obj == undefined) {
    return;
  }
  else {

  for (var k in obj) {

    if (typeof obj[k] === 'string') { // se è una stringa
      // controllo tramite regExp il tipo della stringa

      ...

      if (pos === 0) { // è una e-mail
        label = "<label class='json-label'>"+k+": <input type='text' class='
    json-data-email edit' id='"+path+k+"' value='"+obj[k]+"' ></label><br>";
      }
      else if (pos === 1) { // è una URL

        if ( regExpArray[4].test(obj[k]) ) { // se true è un'immagine
          label = "<label class='json-label'>"+k+": <input type='file' class=
    'json-data-img edit' id='"+path+k+"' ></label><br>";
        }
        else {
          label = "<label class='json-label'>"+k+": <input type='text' class=
    'json-data-url edit' id='"+path+k+"' value='"+obj[k]+"' ></label><br>";
        }
      }
      else if (pos === 2 || pos === 3) { // è un colore
        label =  "<label class='json-label'>"+k+": <span class='pick-color'>
    pick a color >> <input class='json-data-color jscolor edit' id='"+path+k
    +"' value='"+obj[k]+"'></span></label><br>";
      }
      else{ // è una normale stringa
        // costruzione input-text o text-area
        ...
      }
    }

    else if (typeof obj[k] === 'number') { // se è una numero
      var label = "<label class='json-label'>"+k+": <input type='number'
    class='json-data-number edit' id='"+path+k+"' value='"+obj[k]+"' ></label
    ><br>";
      $(el).append(label);

    }

    ...

    // individuazione altri tipi e ricorsione per Array e Object

    ...
  }
}
```

Codice 7.6: Estratto della funzione `parse()`

7.4.2 Modifica dei dati

La funzione `creaTml()` oltre ad occuparsi di creare l'editor giusto per ogni template, si occupa di aggiornare i dati in seguito a una modifica.
La modifica del model del template selezionato avviene tramite l'utilizzo del metodo `Ractive.set()`, offerto dalla libreria *Ractive.js*.
Questo metodo modifica il model ed in seguito lancia l'evento *update* che comporta il re-rendering del template nella pagina, permettendo la visualizzazione delle modifiche in tempo reale.

7.4.3 Controllo degli input

Prima di invocare la modifica di un dato, in tutti e due i tipi di editor, viene effettuato un controllo sul dato inserito per verificarne la conformità con il tipo richiesto.
Per quanto riguarda i tipi non primitivi, citati in precedenza, vengono utilizzate le espressioni regolari per effettuare il controllo.
Nel caso in cui i dati inseriti non siano corretti viene visualizzato un messaggio di errore tramite un *alert-box*.

Capitolo 8

Conclusioni

In questo capitolo finale vengono tratte le conclusioni riguardo alle attività svolte durante il periodo di stage.

8.1 Valutazione del risultato e di Ractive.js

8.1.1 Requisiti soddisfatti

8.2 Criticità

8.3 Conoscenze acquisite