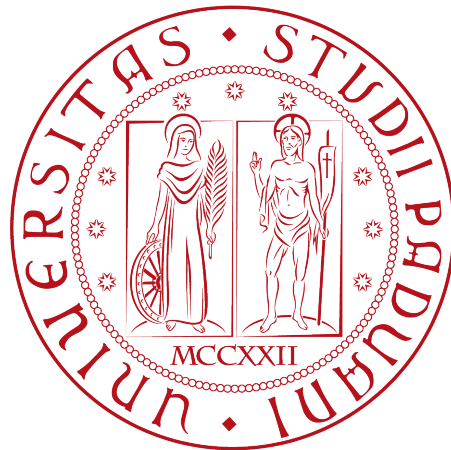


Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA

CORSO DI LAUREA IN INFORMATICA



Sviluppo di applicazioni web in Material Design con AngularJS

Tesi di laurea triennale

Relatore

Prof. Paolo Baldan

Laureando

Andrea Ongaro

ANNO ACCADEMICO 2015-2016

Sommario

In questo documento si descrive il lavoro del laureando Andrea Ongaro, durante il periodo di stage presso l'azienda Wannaup S.r.l.s.. Lo stage era rivolto allo sviluppo del front-end di applicazioni web, mediante l'utilizzo di HTML5, CSS3 e JavaScript, utilizzando il framework AngularJS. In particolare, questo documento tratterà dello sviluppo di uno dei prodotti dell'azienda, l'applicazione web Pollit. Pollit si divide in un sistema per la raccolta di feedback da parte dei clienti, che vengono remunerati per la partecipazione a questionari di campagne commerciali e in una dashboard con analitiche per il committente che ha costantemente sotto controllo l'andamento delle proprie campagne.

Ringraziamenti

Inizio ringraziando il relatore della mia tesi, il Professor Paolo Baldan, per l'aiuto che mi ha fornito nella stesura di questo documento.

Voglio ringraziare in modo particolare Wannaup S.r.l.s. ed il mio tutor Pietro per la stupenda esperienza di stage, per avermi insegnato molto e per avermi dato la possibilità di utilizzare tecnologie fresche ed importanti.

Immane è il ringraziamento pieno d'affetto alla mia famiglia, che mi ha sempre sostenuto nel mio percorso di studi.

Non posso non menzionare gli amici, mio fratello Giacomo e i compagni di corso che hanno reso l'esperienza dell'università unica. Senza di loro non avrei raggiunto questo traguardo.

Infine desidero ringraziare Alessandra per avermi sempre incoraggiato, per avermi sopportato e per avermi fatto crescere.

Padova, Dicembre 2015

Andrea Ongaro

Indice

1	Introduzione	1
1.1	L'azienda	1
1.2	Il progetto	1
1.3	Obiettivi dello stage	2
1.4	Criticità e difficoltà incontrate	3
1.5	Soluzione prodotta	3
2	Strumenti e tecnologie utilizzati	5
2.1	AngularJS	5
2.1.1	Data-binding, View e Model	5
2.1.2	Dependency injection	6
2.1.3	I moduli	7
2.1.4	\$scope	8
2.1.5	I controller	9
2.1.6	Le directive	10
2.1.7	I Service	12
2.2	Material Design	13
2.3	Angular Material	14
2.3.1	Vantaggi e svantaggi della libreria	15
2.4	Bower	16
2.5	JWT Authentication	17
2.6	Atom	17
2.7	Google Chrome Dev Tools	18
3	Analisi dei Requisiti	19
3.1	Classificazione dei requisiti	19
3.2	Descrizione generale dei requisiti di Pollit	20
3.3	Requisiti di pollcenter	21
3.3.1	Requisiti Funzionali	22
3.3.2	Requisiti di Vincolo	24
3.3.3	Riepilogo requisiti	24
3.4	Requisiti di pollboy	25
3.4.1	Requisiti Funzionali	26
3.4.2	Requisiti di Vincolo	27
3.4.3	Riepilogo requisiti	27
4	Progettazione	29
4.1	Considerazioni generali	29

4.1.1	Suddivisione in moduli	29
4.1.2	Utilizzo dei controller e dei service	31
4.2	API di Pollit	32
4.2.1	Campagne	32
4.2.2	Poll	34
4.2.3	Kpi	36
4.2.4	URL	37
4.3	pollcenter	38
4.3.1	Modulo login	40
4.3.2	Modulo common	41
4.3.3	Modulo campaigns	41
4.3.4	Modulo polls	44
4.3.5	Modulo kpis	45
4.3.6	Modulo urls	45
4.4	pollboy	47
4.4.1	Modulo login	48
4.4.2	Modulo campaigns	48
4.4.3	Modulo polls	49
5	Realizzazione	51
5.1	pollcenter	51
5.1.1	Realizzazione dell'autenticazione alle API con JWT	51
5.1.2	Routing dell'applicazione	52
5.1.3	Campagne	54
5.1.4	Poll	57
5.1.5	Kpi	61
5.1.6	URL	61
5.2	pollboy	62
6	Conclusioni	65
6.1	Valutazione dei risultati ottenuti	65
6.2	Aspetti critici nell'uso framework AngularJS	65
6.3	Aspetti critici nell'uso del Material Design	66
6.4	Conoscenze acquisite	67
	Glossary	69
	Riferimenti	70

Elenco delle figure

1.1	Logo di Wannau S.r.l.s.	1
2.1	Two-way data-binding di AngularJS	6
2.2	Un esempio di FAB button	14
2.3	Esempi di card	15
2.4	Card con Angular Material	15
3.1	Esempio di dashboard che mostra le campagne create mediante card	21
4.1	Layout della pagina di login a <i>pollcenter</i>	40
4.2	Card che espone le statistiche di campagne, poll e kpi	41
4.3	Dashboard in cui ogni card rappresenta una campagna. Utilizzando il FAB si possono aggiungere campagne	42
4.4	Una campagna nel dettaglio con i poll ad essa associati rappresentati dalle card	43
4.5	Esempio grafico di una domanda, con inserimento del titolo e aggiunta dei tag che rappresentano i kpi mediante autocompletamento	44
4.6	Esempio grafico di un URL per una campagna. Attraverso il FAB sarà possibile aggiungerne di nuovi	46
4.7	Esempio grafico dell'interfaccia di pollboy su smartphone. Sarà possibile esprimere la preferenza mediante gesture	47
5.1	View della dashboard delle campagne di <i>pollcenter</i>	56
5.2	View di dettaglio e di modifica di una campagna	57
5.3	View di dettaglio e di modifica di una poll. Il FAB consente di aggiungere una domanda tra i tre tipi disponibili	58
5.4	La directive per le domande di tipo rating compilata nella view	58
5.5	La directive per le domande di tipo scelta testuale compilata nella view	59
5.6	La directive per le domande di tipo scelta di immagini compilata nella view	59
5.7	La creazione di un URL dall'applicazione	61
5.8	Una domanda a scelta testuale nell'interfaccia di pollboy	62
5.9	Una domanda a scelta di immagini nell'interfaccia di pollboy	63

6.1	<i>pollcenter</i> su smartphone	66
-----	---	----

Elenco delle tabelle

3.1	Requisiti Funzionali di pollcenter	23
3.2	Requisiti di Vincolo per pollcenter	24
3.3	Numero di requisiti per importanza	24
3.4	Numero di requisiti per tipologia	24
3.5	Requisiti Funzionali di pollboy	26
3.6	Requisiti di Vincolo per pollboy	27
3.7	Numero di requisiti per importanza	27
3.8	Numero di requisiti per tipologia	27

Elenco dei frammenti di codice

2.1	Primo esempio di applicazione in AngularJS	5
2.2	Inline array annotation per la dependency injection	7
2.3	Esempio modulo	7
2.4	Esempio di scope annidati	8
2.5	Il contenuto della view a seguito della compilazione	8
2.6	Esempio di creazione di un controller	9
2.7	Esempio di sintassi controller as	9
2.8	Matching di una directive via attributo	10
2.9	Matching di una directive via classe	10
2.10	Matching di una directive via elemento	10
2.11	Matching di una directive via commento	11
2.12	Esempio di directive definita dall'utente	11
2.13	Esempio di dichiarazione di una directive definita dall'utente	12
2.14	Integrazione di Angular Material in un'applicazione in AngularJS	14
2.15	Esempio di directive di Angular Material	14
2.16	Esempio file bower	16
2.17	Esempio di header JWT	17
2.18	Esempio di payload JWT	17
2.19	Esempio di firma JWT	17

4.1	Esempio di suddivisione in directory per componenti di un'applicazione	29
4.2	Esempio di suddivisione in moduli di un'applicazione	30
4.3	Suddivisione in directory adottata	30
4.4	JSON schema dell'API GET api/campaigns	32
4.5	JSON schema dell'API GET api/campaigns/campaignId	32
4.6	JSON schema dell'API POST api/campaigns/	33
4.7	JSON schema dell'API GET api/campaigns/campaignId/stats/votesovertime	33
4.8	JSON schema dell'API GET api/campaigns/urlH	34
4.9	JSON schema dell'API post api/campaigns/campaignId/vote	34
4.10	JSON schema dell'API GET api/campaigns/campaignId/polls	34
4.11	JSON schema dell'API POST api/campaigns/campaignId/polls	36
4.12	JSON schema dell'API GET api/campaigns/campaignId/polls/pollId/-stats/votesvertime	36
4.13	JSON schema dell'API GET api/kpis	36
4.14	JSON schema dell'API POST api/kpis/	37
4.15	JSON schema dell'API GET api/kpis	37
4.16	JSON schema dell'API POST api/urlhandles/	38
4.17	Struttura di pollcenter	39
4.18	Struttura di pollboy	47
5.1	Realizzazione dell'autenticazione JWT	51
5.2	Realizzazione dell'autenticazione JWT con interceptor per errore 401	52
5.3	Routing di pollcenter	52
5.4	Template della view campaigns.html che verrà compilata a seguito del routing	53
5.5	Esempio di richiesta HTTP	54
5.6	Implementazione di campaignsService	54
5.7	Upload di immagini	60
5.8	Implementazione della view di visualizzazione e modifica di un poll	60

Capitolo 1

Introduzione

1.1 L'azienda

Wannaup S.r.l.s. è una startup avviata da due anni dai soci Lorenzo Gambato e Pietro De Caro. L'azienda opera nell'ambito del **B2B**, offrendo servizi in cloud e sfruttandone l'alta scalabilità come vantaggio competitivo. In particolare Wannaup ha prodotto sistemi gestionali in cloud, webapp, strumenti per business intelligence, sistemi di raccomandazione, **DAM** (Digital Asset Management) e siti web di tipo e-commerce. Oltre a questi prodotti realizzati su commessa, l'azienda sta realizzando alcuni prodotti proprietari e proprio uno di questi rappresenta il tema principale di questa tesi.



Figura 1.1: Logo di Wannaup S.r.l.s.

1.2 Il progetto

Nel corso dello stage sono stati affrontati più temi e lo studente ha collaborato alla realizzazione di due prodotti. In questo documento verrà presentato solamente uno dei progetti, in quanto le tempistiche dello stage e le priorità aziendali hanno portato lo studente a concentrarsi maggiormente sull'applicativo denominato **Pollit**.

Pollit è una webapp con funzionalità **CRM** (Customer Relationship Management) e customer satisfaction, che permette agli utenti che partecipano ad eventi o che si trovano in punti vendita di rispondere ad una serie di domande poste da un brand, ricevendo una sorta di premio al termine della partecipazione al questionario completo (ad esempio un coupon o un buono sconto). Il reparto marketing del brand ha a disposizione una dashboard attraverso cui analizzare in profondità e segmentare i dati raccolti, potendoli poi riassumere in specifici indicatori di performance globali. Si è quindi deciso di puntare a realizzare due applicazioni web distinte che formano il prodotto **Pollit**: una che consenta all'utente di partecipare ad un sondaggio, denominata *pollboy* e una dashboard per il committente che vuole raccogliere dati sui sondaggi e sulle campagne da lui create, denominata *pollcenter*.

Il progetto riguarda quindi la progettazione delle UI (User Interface) dei due applicativi e la realizzazione degli stessi. L'azienda ha deciso di prestare particolare attenzione al design dell'interfaccia utente, che dovrà possedere caratteristiche di usabilità, rendere accattivante e semplice partecipare ai poll e consentire agli utenti della dashboard di creare campagne e consultare le analitiche in modo semplice ed efficace.

L'azienda e lo studente hanno individuato alcune caratteristiche che il prodotto dovrà avere. Esse sono espresse in termini di usabilità delle interfacce, di scalabilità degli applicativi e di utilizzo di tecnologie moderne e consolidate e sono le seguenti:

- utilizzo del framework AngularJS;
- utilizzo di CSS3 e HTML5;
- gli applicativi dovranno essere responsive e adattarsi sia in ambiente mobile che in ambiente desktop, senza pregiudicare l'esperienza utente;
- la progettazione della User Interface verrà effettuata prendendo in considerazione il Material Design di Google;

Questi vincoli verranno espressi successivamente in termini di requisiti nella sezione riguardante [Analisi dei Requisiti](#).

I punti focali nella realizzazione del prodotto si possono riassumere in esigenze di design e tecnologiche. Ma separarle nettamente non sarebbe totalmente corretto, poiché sia lo studio di interfacce grafiche che la realizzazione e la scelta delle tecnologie possono condizionarsi l'un l'altra. Ad esempio la progettazione architetturale degli applicativi è stata influenzata dalle scelte stilistiche, di usabilità e della [UX](#) (User eXperience) con cui si è voluto realizzare **Pollit**, producendo una struttura per le applicazioni non [MVC](#) ma basata sulla suddivisione per funzionalità dei moduli (decisione documentata e spiegata nel dettaglio nel capitolo [Progettazione](#)). Si può notare come l'azienda punti molto all'ottenimento di un prodotto graficamente accattivante, semplice nell'utilizzo e che coinvolga in un'esperienza piacevole il customer (utente finale che partecipa ai sondaggi), contrariamente ad altre metodologie di somministrare questionari online che rendono snervante il loro completamento. Dal punto di vista tecnologico si ricercano strumenti e linguaggi recenti, che consentano di ottenere caratteristiche di scalabilità, performance, riuso e innovazione.

1.3 Obiettivi dello stage

L'obiettivo dello stage è quello di produrre un **prototipo** che presenti le principali funzionalità di **Pollit**, secondo i requisiti espressi nel capitolo [Analisi dei Requisiti](#).

All'interno del progetto **Pollit**, lo stage si è incentrato nella progettazione delle interfacce grafiche utente, nella revisione dell'analisi dei requisiti svolta in precedenza dall'azienda, nella progettazione architetturale e nello sviluppo del front-end dei due applicativi. Il back-end invece è stato in precedenza realizzato dall'azienda, che rende possibile interfacciarsi per la lettura e la modifica dei dati mediante l'esposizione di [API](#) che seguono il principio architetturale [REST-Like](#).

I test sulle componenti delle applicazioni non sono stati effettuati dallo studente, che invece ne ha sviluppato il codice, ma da un team apposito e da strumenti automatici.

1.4 Criticità e difficoltà incontrate

La principale difficoltà pervenuta nello svolgimento del progetto è stata l'apprendimento del framework AngularJS. Il framework fornisce strumenti e tecniche potenti ed efficaci rispetto ad una programmazione puramente in JavaScript, ma introduce concetti e design pattern che necessitano una comprensione profonda prima di poter essere applicati propriamente. Le pratiche di sviluppo, le best-practices previste e gli elementi più complessi di Angular hanno reso necessario un breve periodo di studio, che ha visto il coinvolgimento del tutor aziendale.

Proprio a causa della iniziale complessità del framework, alcuni componenti sono stati ricodificati in passaggi successivi al loro completamento per ottimizzare il codice dal punto di vista delle prestazioni e del flusso di esecuzione.

L'integrazione con il team e l'adeguamento al metodo di lavoro dell'azienda non si sono rivelati problematici o bloccanti, anche grazie all'utilizzo di strumenti per lo sviluppo e l'organizzazione già impiegati e sfruttati in esperienze precedenti.

1.5 Soluzione prodotta

Al termine dello stage è stato possibile completare i prototipi degli applicativi, in modo che rispettassero tutti i requisiti individuati. Purtroppo il tempo limitato dello stage non ha permesso di apportare ulteriori miglioramenti e valutare funzionalità aggiuntive. I prototipi sono stati prodotti utilizzando completamente le linee guida del Material Design e possiedono quindi un layout che ne consente il completo utilizzo sia in ambiente desktop che mobile. Alcuni screenshot dei risultati sono disponibili nel capitolo [Realizzazione](#).

Capitolo 2

Strumenti e tecnologie utilizzati

Questo capitolo descrivono le tecnologie impiegate nella realizzazione di **Pollit**, mostrandone pregi e difetti e giustificandone la scelta. Verranno quindi proposti il framework AngularJS e le linee guida del Material Design di Google. Si è scelto di non trattare HTML5 e CSS3, ma è possibile trovare fonti e spunti nei riferimenti bibliografici.

2.1 AngularJS

AngularJS è un framework per il linguaggio JavaScript per la creazione di applicazioni web dinamiche. Esso consente di estendere la normale sintassi dell'HTML, per manipolare dinamicamente il comportamento di un elemento, attraverso l'utilizzo di costrutti denominati *directive* che verranno trattati in seguito.

AngularJS è un framework strutturale che impiega un pattern derivante da [MVC](#): *MVW* ovvero *Model View Whatever*. Definirlo propriamente un design pattern sarebbe errato, poiché *Whatever* significa in questo contesto “qualunque cosa funzioni per te”. È infatti possibile strutturare la propria applicazione in modo flessibile, scegliendo di applicare approcci più rigidi come [MVC](#) o variare secondo le proprie necessità di flessibilità.

Le caratteristiche di AngularJS verranno ora mostrate. Ovviamente non è possibile descrivere totalmente le funzionalità del framework. Si è quindi voluto elencare in generale quali sono i vantaggi, gli svantaggi, le componenti ed i pattern offerti.

2.1.1 Data-binding, View e Model

Per come è possibile strutturare un'applicazione in AngularJS è necessario capire il data-binding che il framework offre, chiamato **two-way data-binding**. Il funzionamento del framework è illustrato in questo esempio:

```
1 <div ng-app ng-init="qty=1;cost=2">
2   <b>Ordine:</b>
3   <div>
4     Quantita: <input type="number" min="0" ng-model="qty">
5   </div>
6   <div>
7     Costo: <input type="number" min="0" ng-model="cost">
8   </div>
9 </div>
```

```

10 <b>Totale:</b> {{qty * cost}}
11 </div>
12 </div>

```

Codice 2.1: Primo esempio di applicazione in AngularJS

Questo codice HTML viene denominato in Angular *template*. All'avvio dell'applicazione, il compilatore del framework effettua il parsing del template, e carica il [DOM](#) della pagina e la renderizza. Questo prodotto della compilazione rappresenta una *view*.

Si può notare che il markup degli elementi HTML è stato esteso da alcune directives. Ad esempio *ng-app*, che inizializza l'applicazione ed *ng-model* che in questo caso consente all'input di leggere e modificare il valore di *cost*. Inizialmente, quando viene renderizzata la *view*, l'elemento input avrà come valore il valore iniziale di *cost*, ovvero 2, e potrà modificarlo ripercuotendo la modifica sulla variabile stessa.

Un altro tipo di markup introdotto da Angular è `{{ qty * cost }}`. Quando il compilatore incontra il markup `{{ expression }}`, lo rimpiazza con il valore dell'espressione scritta in sintassi JavaScript.

La directive *ng-model* vista in precedenza, realizza il data-binding fra *view* e *model*. Per un'applicazione in Angular, il data-binding è la sincronizzazione automatica e continua tra *model* e *view*. In questo modo la *view* è trattata come una proiezione del *model* e, se esso cambia, i cambiamenti vengono riflessi sulla *view* e viceversa.

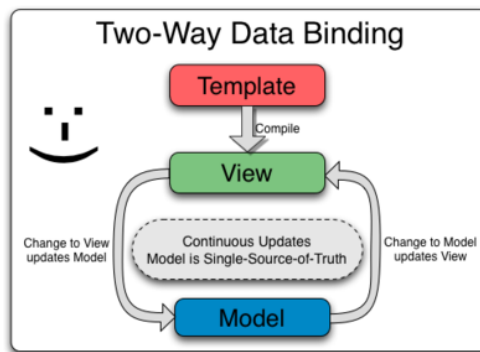


Figura 2.1: Two-way data-binding di AngularJS

La figura mostra il funzionamento del two-way data-binding. Prima il template (ovvero il markup HTML arricchito dal markup delle directive) viene compilato nel browser, producendo una *view*. Ogni cambiamento apportato nella *view* viene riflesso sul *model* e viceversa ogni cambiamento del *model* (da parte di qualunque altro componente che ne abbia accesso) si propaga alla *view*. Poiché la *view* è una semplice proiezione dello stato del *model*, il controller viene completamente separato da essa ed è inconsapevole di come essa manipola il *model*. Ciò si traduce in una **riduzione delle dipendenze** dell'applicazione e consente di effettuare test del controller in isolamento dal [DOM](#).

2.1.2 Dependency injection

Lasciare al componente il compito di risolvere le proprie dipendenze, creando gli oggetti necessari al suo funzionamento, aumenta l'accoppiamento tra le componenti e rende più

difficoltoso progettare i test di unità. Con questo pattern invece è possibile esprimere le dipendenze in modo dichiarativo e utilizzare un oggetto contenitore per risolverle dinamicamente a runtime. In questo modo è possibile scegliere anche quale componente iniettare in base allo stato del programma.

I componenti coinvolti nel Dependency Injection sono:

- un client che viene creato e riceve le dipendenze;
- un contenitore che si occupa di creare il client e di iniettarvi le dipendenze;
- un servizio che deve essere iniettato al client;

Nello specifico di AngularJS, il componente `$injector` funziona da contenitore che si occupa di risolvere le dipendenze.

Angular consente al programmatore diversi modi di dichiarare le dipendenze. Nel caso specifico del progetto di stage è stato usato l'approccio *inline array annotation*: quando si vuole creare un controller o un altro componente, ad esso viene passato un array contenente la lista delle componenti con cui ha dipendenze e una funzione che costruisce il componente stesso e che viene dichiarata con gli stessi parametri della lista di dipendenze.

```
1 someModule.controller('MyController', ['$scope', 'greeter', function($scope,  
2   // ...  
3   }]);
```

Codice 2.2: Inline array annotation per la dependency injection

A differenza di altre tecniche, l'inline array annotation consente di poter effettuare una [minificazione](#) del codice in fase di rilascio dell'applicativo.

2.1.3 I moduli

Un modulo per AngularJS è visibile come un contenitore per le differenti parti dell'applicazione, ad esempio directive, controller, service, ecc. Poiché le applicazioni di Angular non possiedono un *main*, come applicazioni in altri linguaggi, un modulo dichiara come deve avvenire il bootstrap dell'applicazione. Ad esempio:

```
1 angular.module('simpleModule', ['moduleA', 'moduleB'])  
2 ...
```

Codice 2.3: Esempio modulo

indica che deve essere istanziato un modulo chiamato *simpleModule*. Esso fa utilizzo (e quindi ha dipendenze) dei moduli *moduleA* e *moduleB* che quindi devono essere caricati nell'applicazione prima di *simpleModule* e nell'ordine della lista.

Organizzare l'applicazione in moduli presenta alcuni vantaggi:

- la sintassi dichiarativa delle dipendenze è semplice da gestire;
- è possibile suddividere il codice in moduli riusabili da altre applicazioni;
- è possibile isolare i singoli moduli per l'esecuzione di test di unità.

Tipicamente è possibile utilizzare moduli di terze parti includendoli semplicemente nella lista delle dipendenze.

2.1.4 \$scope

In AngularJS il concetto di **scope** ha particolare importanza. Lo scope è un oggetto che possiede un riferimento al model dell'applicazione. Gli scope possono essere annidati per limitare l'accesso a particolari componenti dell'applicativo e per dare accesso a proprietà condivise del model. Gli scope annidati possono essere definiti *child scope* se ereditano proprietà dallo scope padre, oppure *scope isolati* utilizzati da directive definite dall'utente e descritte nelle sezioni successive. Per ogni applicazione, AngularJS crea in automatico uno scope radice di quelli presenti nell'applicazione, detto *root scope*.

Lo scope è il collante tra controller di un'applicazione e le corrispondenti view. Infatti sia una view che il controller associato possiedono un riferimento allo scope ma né il controller ha riferimento alla view, né viceversa. Questo consente di realizzare controller isolati e indipendenti sia dalle view che dal **DOM**.

```

1 <div class="show-scope-demo">
2   <div ng-controller="GreetController">
3     </div>
4     <div ng-controller="ListController">
5       Hello {{name}}!
6       <ol>
7         <li ng-repeat="name in names">{{name}} from {{department}}</li>
8       </ol>
9     </div>
10  </div>
11
12 angular.module('scopeExample', [])
13 .controller('GreetController', ['$scope', '$rootScope', function($scope,
14   $rootScope) {
15   $scope.name = 'World';
16   $rootScope.department = 'Computer Science';
17 }])
18 .controller('ListController', ['$scope', function($scope) {
19   $scope.names = ['Igor', 'Misko', 'Vojta'];
20 }]);

```

Codice 2.4: Esempio di scope annidati

In questo esempio sono stati creati due controller, associati al template (e quindi successivamente alla compilazione, alle view) che utilizzano il riferimento allo scope per aggiungere delle proprietà al model. In questo caso, viene creato uno scope padre con proprietà `name = 'World'` e viene aggiunto al root scope `departement = 'Computer Science'`. All'interno dello scope in cui è stato inserito in controller *ListController*, il compilatore incontra l'espressione `{ {name} }`. Inizialmente cerca la proprietà `name` nello scope del controller e se non la trova, come in questo caso, la ricerca nello scope padre, risalendo eventualmente la gerarchia fino a raggiungere il root scope. Quindi, l'espressione dell'esempio viene valutata e il compilatore stampa nella view "World".

Successivamente la directive *ng-repeat* crea elementi in una lista HTML per ogni elemento dell'array e `{ {departement} }` legge dal root scope il valore "Computer Science". Il risultato finale è:

```

1 Hello World!
2   1. Igor from Computer Science
3   2. Misko from Computer Science
4   3. Vojta from Computer Science

```

Codice 2.5: Il contenuto della view a seguito della compilazione

2.1.5 I controller

In AngularJS un controller viene definito da una funzione costruttore che viene utilizzata per aumentare lo scope.

```
1 myApp.controller('GreetingController', ['$scope', 'moduleA', function($scope,
2   moduleA) {
3     $scope.greeting = 'Hola!';
4   }]);
5 <div ng-controller="GreetingController">
6   {{ greeting }}
7 </div>
```

Codice 2.6: Esempio di creazione di un controller

Quando un controller viene aggiunto al [DOM](#) mediante la directive *ng-controller*, il framework istanzia un nuovo oggetto controller utilizzando il suo costruttore e crea un nuovo child scope che attraverso la dependency injection sarà disponibile al controller. Buona prassi è usare un controller in AngularJS per impostare uno stato iniziale dello scope associato, per variare il comportamento o i dati nello scope e per eseguire solo la business logic necessaria al funzionamento di una singola view. Tipicamente associare un controller a più view può essere difficoltoso da gestire e fonte di errori. La documentazione di Angular sconsiglia di usare controller per:

- manipolare il [DOM](#) (quindi promuove l'indipendenza e l'isolamento dei controller);
- condividere codice stati con altri controller (possono essere usati i *services*, descritti in seguito);
- gestire il ciclo di vita di altri componenti, ad esempio di altri controller.

I controller in Pollit

All'interno del progetto **Pollit** è stato deciso di scrivere i controller mediante la sintassi *controller as*:

```
1 myApp.controller('ParentCtrl', ['$scope', 'moduleA', function($scope, moduleA
2   ) {
3     this.name = 'Carlo';
4   }]);
5 myApp.controller('ChildCtrl', ['moduleA', function(moduleA) {
6     this.name = 'Filippo';
7   }]);
8
9 <body ng-controller="ParentCtrl as parent">
10   <input ng-model="parent.name" /> {{parent.name}}
11
12   <div ng-controller="ChildCtrl as child">
13     <input ng-model="child.name" /> {{child.name}} - {{parent.name}}
14   </div>
15 </body>
```

Codice 2.7: Esempio di sintassi controller as

In questo caso l'istanza del controller viene assegnata ad una proprietà dell'oggetto scope (*parent* nel primo caso, *child* nel secondo).

L'impiego di questa tecnica apporta alcuni benefici:

- è possibile annidare controller, mantenendo una sintassi sempre molto leggibile;
- è possibile utilizzare lo stesso nome per proprietà di controller differenti;
- non è necessario iniettare la dipendenza dello scope, ma è comunque possibile farlo;
- è possibile utilizzare il controller come un vero e proprio oggetto JavaScript, non solo come un modo di aumentare le proprietà di uno scope.

2.1.6 Le directive

Le directive sono attributi, elementi o classi che individuano un elemento del **DOM** e che segnalano al compilatore HTML di AngularJS di aggiungere un particolare comportamento a quell'elemento.

Angular fornisce alcune directive predefinite come: *ngModel*, che consente il two-way data-binding tra una proprietà del model e un elemento del **DOM**; *ngClass*, che consente di cambiare classe dinamicamente ad un elemento del **DOM**; *ngController* che dichiara lo spazio di un controller all'interno del **DOM** e molte altre.

Esistono diverse soluzioni per effettuare un matching tra un elemento e una directive:

- **via attributo:** il compilatore aggiunge un comportamento alla directive mediante il riconoscimento di un attributo nel markup dell'elemento del **DOM**.

```
1 <input ng-model="foo">
```

Codice 2.8: Matching di una directive via attributo

In questo esempio l'elemento input corrisponde alla directive *ngModel*;

- **via classe:** in questo caso il compilatore riconosce la directive dichiarata nell'attributo `class` dell'elemento del **DOM**.

```
1 <div class="my-dir"></div>
```

Codice 2.9: Matching di una directive via classe

In questo caso il compilatore aggiunge il comportamento della directive *myDir*;

- **via elemento:** è possibile dichiarare un elemento del **DOM** con il nome di una directive per far sì che il compilatore ne sostituisca il comportamento.

```
1 <person>{{name}}></person>
```

Codice 2.10: Matching di una directive via elemento

Il matching nell'esempio avviene tra elemento *person* e directive *person*.

- **via commento:** le directive possono essere riconosciute anche quando vengono esplicitate in un commento HTML

```
1 <!-- directive: my-dir -->
```

Codice 2.11: Matching di una directive via commento

Tuttavia le best practices del framework consigliano soltanto l'utilizzo delle directive mediante il nome del tag o l'attributo di un elemento per facilitare la lettura del codice al programmatore e semplificare il riconoscimento della correlazione fra elemento e directive.

È importante considerare che il framework fa utilizzo della notazione camel case per i nomi delle directive. Infatti ogni directive, sia fornita da Angular che definita dallo sviluppatore (nella sezione successiva), deve avere un nome in camel case, ad esempio “myDir”, che nel markup del **DOM** si deve tradurre mediante il simbolo “-” : “my-dir”. Questa notazione si estende anche ad eventuali attributi della directive.

Directive definite dall'utente

Come per i controller, le directive definite dall'utente vengono registrate all'interno di un modulo dell'applicazione.

```
1 angular.module('simpleModule', [])
2 .directive('myCustomer', ['moduleA', function(modA) {
3   return {
4     templateUrl: '/path/to/template',
5     restrict: 'E',
6     scope: {
7       name: '@',
8       customer: '=',
9       aFunction: '&'
10    },
11    controller: function($scope, element, attrs){
12      //do something
13    }
14  };
15 }]);
```

Codice 2.12: Esempio di directive definita dall'utente

La directive viene creata mediante una funzione factory e contiene le proprietà:

- **templateUrl:** opzionale, è l'indirizzo al quale il compilatore può trovare il template della directive, ovvero un template HTML che esso sostituirà nel **DOM** quando incontra il markup della directive;
- **restrict:** opzionale, indica se il matching della directive deve avvenire soltanto mediante nome dell'attributo (*A*), nome dell'elemento, ovvero tag, (*E*), nome della classe (*C*) oppure sia classe che elemento o attributo (*AEC*). La restrizione di default è per elemento o attributo;
- **scope:** con la directive, Angular crea uno scope isolato associato ad essa. Deve essere dichiarato come un oggetto JavaScript ed ogni proprietà avrà un valore associatogli dal compilatore quando avverrà la sostituzione della directive nel **DOM**. Le proprietà vengono rappresentate nel **DOM** con il markup di attributo. Ad esempio, la directive *myCustomer* può essere dichiarata come:

```

1 <div ng-controller="MyController as myCtrl">
2   <my-customer name="Tizio" customer="myCtrl.aCustomer" a-function="
   myCtrl.simpleFunc()"></my-customer>
3 </div>

```

Codice 2.13: Esempio di dichiarazione di una directive definita dall'utente

Ad ogni proprietà dello scope della directive viene associato un binding con le proprietà con cui gli attributi vengono dichiarati, con dei vincoli:

- @: indica che il valore passato all'attributo sarà visibile nello scope della directive mediante la proprietà dichiarata con @;
 - &: consente allo scope isolato della directive di eseguire e valutare un'espressione (tipicamente una funzione) presente nel contesto dello scope padre, con la possibilità di passare dei parametri. Nell'esempio è possibile per lo scope della directive eseguire l'istruzione `$scope.aFunction()`, che causerà la chiamata e la risoluzione della funzione con cui l'attributo ha un binding;
 - ==: effettua un two-way data-binding tra lo scope isolato della directive e lo scope padre mediante un attributo. Nell'esempio precedente viene effettuato un binding tra la proprietà *customer* della directive e *aCustomer* presente nello scope del controller. In questo caso ogni modifica di quest'oggetto nello scope della directive verrà riflessa nello scope del controller e viceversa.
- **controller:** è possibile associare un controller allo scope isolato della directive, che consente quindi di manipolarlo e di eseguire operazioni più complesse. Il controller è dichiarato come una funzione a cui è possibile passare come parametri lo scope della funzione, l'eventuale elemento a cui la directive è stata sostituita ed i suoi attributi. È possibile quindi eseguire codice all'interno della directive utilizzando un controller. Vi è però una seconda possibilità, ovvero utilizzare una funzione *link*. Le differenze tra i due approcci sono:
 - se viene dichiarato un controller, il codice viene eseguito prima che intervenga il compilatore di Angular;
 - se viene dichiarato un link, il codice viene eseguito successivamente alla compilazione della view.

È quindi consigliabile usare un controller quando si vogliono semplicemente eseguire operazioni sullo scope, mentre l'utilizzo di link è migliore nel caso in cui si ha la necessità di manipolare il [DOM](#).

2.1.7 I Service

I service in AngularJS sono degli oggetti utilizzabili per organizzare e condividere codice tra i vari moduli, controller e directive dell'applicazione. I service di Angular sono dei singleton e vengono istanziati solo quando un modulo, un controller o una directive specificano il nome del service nella lista delle dipendenze.

Un tipico esempio di utilizzo dei service avviene quando l'applicazione deve sfruttare le [API](#) di un servizio web: il service si occuperà di effettuare le chiamate [REST](#) agli indirizzi delle [API](#) e fornirà un'interfaccia alle altre componenti per accedere e modificare i dati ottenuti.

2.2 Material Design

Per quanto riguarda la progettazione delle interfacce grafiche, è stato scelto di adottare il Material Design sviluppato da Google, ovvero un insieme di linee guida che fanno uso di layout basati su griglie, animazioni di tipo responsive (che si adattano in base alla dimensione dello schermo), padding ed effetti di profondità.

Il Material Design può essere visto come un insieme di specifiche di visualizzazione, di animazione e di interazione tra componenti e utente che mira ad unificare il design di applicazioni web, portando l'interfaccia ad adattarsi a dispositivi diversi e di dimensione diversa. Questo tipo di design utilizza alcuni canoni dei design flat, aggiungendo agli elementi profondità, consentendo quindi di poter ottenere effetti di luce o di ombra e di sfruttare effettivamente tre dimensioni. Inoltre, Google e Material Design forniscono scale di colori e forme predeterminate da cui attingere, attribuibili agli elementi dell'applicazione in base al loro contesto di utilizzo. Un esempio tipico è rappresentato dai FAB (Floating Action Button): sono bottoni destinati ad innescare un'azione che si vuole promuovere o un'azione principale (ad esempio un bottone che consente di aggiungere un numero ad una rubrica di contatti telefonici). Secondo le linee guida essi devono essere tondi e fluttuanti, con una data distanza dagli angoli dell'interfaccia, possedere profondità ottenuta mediante effetti di ombre, avere dimensioni prestabilite e per ogni vista o pagina dell'applicazione può esserci soltanto un FAB. La scelta di adottare queste linee guida di design è dettata dalle seguenti caratteristiche e **vantaggi** che il Material Design è in grado di dare:

- è in grado di mettere in risalto le interazioni tra l'interfaccia e l'utente, dando enfasi alle azioni (come nel caso dei FAB) che l'utente può compiere;
- fornisce linee guida per quanto riguarda l'esperienza che si desidera far compiere all'utente che utilizza l'applicazione, oltre che estetiche: in questo caso l'interfaccia comunica e guida l'utente;
- l'utente finale ritrova un'interfaccia usata nelle più comuni applicazioni per dispositivi mobili realizzate da Google, a cui è molto probabilmente abituato. Quindi il tempo di apprendimento nell'utilizzo dell'applicativo risulta breve;
- le linee guida e favoriscono la realizzazione di un layout responsive, che si adatta senza sforzi eccessivi alla visualizzazione su dispositivi mobili e desktop.

Sono stati considerati anche dei possibili **svantaggi** nell'impiego del Material Design, ovvero:

- la particolare visualizzazione di elementi di input testuale non si presta bene a form densi di campi da inserire: raggruppandone molti il layout appare confuso e poco attrattivo per l'utente;
- il tentativo di dare profondità agli elementi e il seguire le animazioni consigliate comporta un più lento caricamento della pagina: sono necessari fogli di stile complessi da applicare per il browser ed è necessario l'impiego di codice JavaScript specifico;
- Material Design è ancora giovane e studiato in particolar modo per l'esperienza su dispositivi mobili, quindi regole rigide su componenti e stili possono rendere difficile sfruttare al meglio lo spazio più ampio su dispositivi desktop.

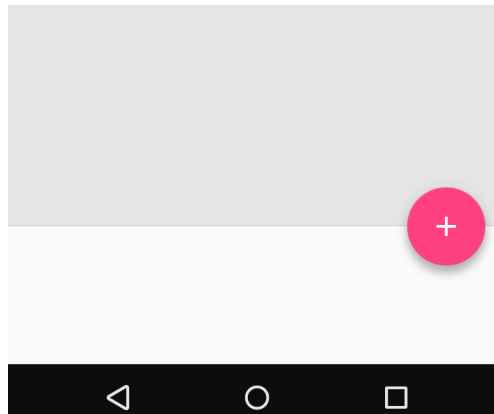


Figura 2.2: Un esempio di FAB button

Poiché **Pollit** deve presentare interfacce semplici, responsive e in particolar modo fruibili su cellulari e tablet, la scelta del Material Design è sembrata opportuna e vantaggiosa.

2.3 Angular Material

Per applicare il Material Design si è scelto di impiegare la libreria Angular Material. Essa implementa le specifiche del design in AngularJS ed è possibile integrarla alla propria applicazione aggiungendo il modulo alla lista dei moduli da iniettare al caricamento:

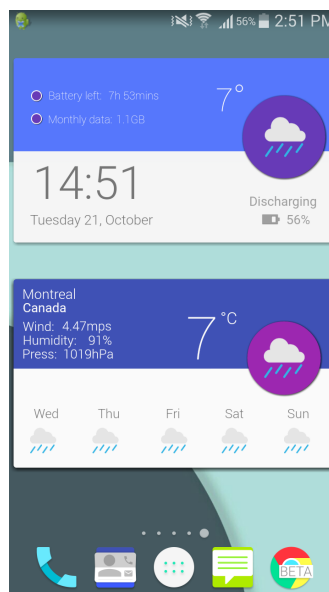
```
1 var app = angular.module('myApp', ['ngMaterial'])
2 ...
```

Codice 2.14: Integrazione di Angular Material in un'applicazione in AngularJS

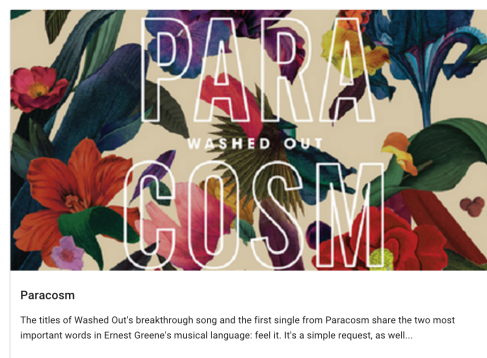
Angular Material offre componenti dell'interfaccia come bottoni, FAB e input testuali, definendone anche gli effetti visuali richiesti dal Material Design. Questi componenti grafici sono utilizzabili all'interno delle view dell'applicazione tramite directive. Ad esempio, supponiamo che si voglia utilizzare una “card”, ovvero un elemento in una pagina che descriva in modo dettagliato una singola fonte informativa e che può contenere testo, foto e link. È possibile utilizzare le directive:

```
1 <md-card>
2   
3   <md-card-content>
4     <h2 class="md-title">Paracosm</h2>
5     <p>
6       The titles of Washed Out's breakthrough song and the first single
7       from Paracosm share the
8       two most important words in Ernest Greene's musical language: feel
9       it. It's a simple request, as well...
10    </p>
11  </md-card-content>
12 </md-card>
```

Codice 2.15: Esempio di directive di Angular Material

**Figura 2.3:** Esempi di card

per produrre il risultato: Un'altra caratteristica di Angular Material è il sistema a

**Figura 2.4:** Card con Angular Material

griglia con cui è possibile ridimensionare e posizionare gli elementi dell'interfaccia, che impiega **flexbox**: utilizzando l'attributo **flex** nel markup di un elemento HTML, e assegnando un valore da 0 a 100, l'elemento verrà automaticamente ridimensionato ad una dimensione percentuale dello spazio disponibile. Ad esempio **flex="30"** causerà il ridimensionamento dell'elemento al 30% dello spazio disponibile nella finestra del browser.

2.3.1 Vantaggi e svantaggi della libreria

La libreria Angular Material presenta dei notevoli **vantaggi** nel contesto di **Pollit**:

- gli elementi del Material Design sono stati definiti e hanno un layout CSS associato;

- gli elementi sono utilizzabili mediante directive;
- il layout a griglia rende semplice l'adattarsi dell'applicazione in contesto desktop e mobile;

Gli **svantaggi** della libreria invece sono i seguenti:

- il layout **flexbox** non viene rappresentato correttamente da tutti i web browser. Tuttavia l'azienda ha deciso di porre come vincolo il funzionamento del prodotto solo sui browser più moderni;
- per applicare le animazioni del Material Design, la libreria fa vasto utilizzo di codice JavaScript, appesantendo quindi il caricamento delle pagine. Poiché **Pollit** presenta un layout molto semplice, si è ritenuto questo problema solo marginale.

2.4 Bower

Bower è un gestore di pacchetti per librerie JavaScript che consente di definire le dipendenze di un'applicazione verso pacchetti di terze parti e di recuperarle tramite il web. È stato utilizzato per le dipendenze dei front-end di *pollcenter* e *pollboy*, per semplificare il versionamento delle applicazioni.

Per utilizzare Bower è necessario che sulla macchina da cui deve essere eseguito sia presente **npm**. È stato definito un file in formato JSON denominato *bower.json* che contiene la lista delle dipendenze dell'applicazione verso moduli di terze parti.

Un esempio di bower file è il seguente:

```
1 //bower.json
2 {
3   "name": "pollcenter",
4   "version": "0.0.0",
5   "description": "The dashboard to manage polls,campaigns and prizes.",
6   "main": "assets/js/app.js",
7   "license": "None",
8   "dependencies": {
9     "angular-ui-router": "~0.2.15",
10    "angular-material": "~0.10.0",
11    "angular": "~1.4.1",
12    "angular-jwt": "~0.0.9",
13    "ng-file-upload": "~5.0.9",
14    "angularjs-nvd3-directives": "~0.0.7",
15    "moment": "~2.10.3"
16  }
```

Codice 2.16: Esempio file bower

Nel file vengono indicate le dipendenze e le versioni minime richieste. Attraverso il comando da terminale **bower install**, dalla directory in cui è presente il file di bower, automaticamente tutte le dipendenze verranno scaricate e verrà creata la cartella **bower_components**, che le contiene.

Se l'applicazione viene mantenuta in un repository per il versionamento, allora è possibile far risiedere in esso soltanto il file di bower. Ogni sviluppatore, una volta clonata la repository, dovrà semplicemente utilizzare il comando **bower_components** per scaricare i file delle dipendenze.

2.5 JWT Authentication

Sia *pollcenter* che *pollboy* applicano un modello di comunicazione fra front-end e back-end dell'applicazione mediante chiamate ad [API](#). Si rende quindi necessario applicare un sistema di sicurezza alle chiamate. In particolare, JWT utilizza dei token in formato JSON per autenticare l'utente che vuole utilizzare le [API](#).

I token JWT sono autocontenuti, ovvero trasportano tutte le informazioni necessarie al loro funzionamento, perciò è molto semplice includerli nell'header di una chiamata HTTP. Ogni token è costituito da tre parti:

- *header*: dichiara il tipo del token, in questo caso JWT e l'algoritmo di codifica che si intende usare

```
1 {  
2   typ: 'JWT',  
3   alg: 'HS256',  
4 }
```

Codice 2.17: Esempio di header JWT

Per essere trasmesso, l'header viene codificato in base64

- *payload*: è il contenuto informativo del token, contiene ad esempio rivendicazioni di permessi di amministratore

```
1 {  
2   "iss": "mysite.com",  
3   "exp": 1300819380,  
4   "name": "Mario Rossi",  
5   "admin": true  
6 }
```

Codice 2.18: Esempio di payload JWT

Come per l'header, anche il payload viene codificato in base64;

- *signature*: è la firma del token e si ottiene codificando in base64 payload e header

```
1 var encodedString = base64UrlEncode(header) + "." + base64UrlEncode(  
  payload);  
2 HMACSHA256(encodedString, 'secret');
```

Codice 2.19: Esempio di firma JWT

secret è la chiave detenuta dal server, che in questo modo può verificare token esistenti e crearne di nuovi.

Il JWT trasmesso nell'header della richiesta HTTP avrà quindi la forma `xxxx.yyyy.zzzz` dove la prima parte è l'header in base64, seguito dal payload ed infine dalla firma.

2.6 Atom

Ovviamente è possibile utilizzare qualsiasi editor di testo per sviluppare applicazioni in JavaScript. L'azienda però ha concordato l'utilizzo dell'editor open source Atom, sviluppato di GitHub. Atom è personalizzabile mediante pacchetti. Alcuni di questi

forniscono autocompletamento per codice JavaScript, HTML, CSS e in particolare per AngularJS.

2.7 Google Chrome Dev Tools

Durante la realizzazione del prodotto si è fatto vasto impiego degli strumenti di sviluppo forniti dal browser Google Chrome. Esso fornisce funzionalità avanzate per il debugging di applicazioni in JavaScript, ad esempio:

- è possibile utilizzare il debugger di Google Chrome per inserire break point all'interno del codice dell'applicativo e quindi interrompere il flusso di esecuzione nei punti desiderati;
- è possibile stampare sulla console di Chrome dal proprio programma mediante l'istruzione `console.log`;
- è possibile verificare l'esito di chiamate ad [API](#) esterne mediante protocollo http;
- è possibile modificare in tempo reale lo stile degli elementi di una pagina;
- è possibile simulare il rendering dell'applicazione su svariati tipi di dispositivi mobili, poiché Chrome ne emula dimensioni, densità di pixel dello schermo e sensori.

Capitolo 3

Analisi dei Requisiti

Il capitolo contiene ed espone i requisiti dei due applicativi che compongono il prodotto **Pollit**.

Il progetto affidato allo studente verteva sulla progettazione e la realizzazione degli applicativi a partire da una analisi dei requisiti precedentemente effettuata dall'azienda. Pertanto non è stato necessario ripartire dai casi d'uso.

Tuttavia lo studente è stato coinvolto in una revisione dei requisiti esistenti e nel loro ampliamento per quanto riguarda le funzionalità di interazione con l'utilizzatore.

3.1 Classificazione dei requisiti

I requisiti individuati sono stati catalogati secondo la seguente codifica:

$$R[T][I][C]$$

dove:

- **Tipo:** specifica la tipologia del requisito e può assumere i valori:
 - **F** - *Funzionale*, determina una funzionalità dell'applicazione;
 - **V** - *Vincolo*, determina un vincolo che il prodotto deve rispettare (tipicamente vincoli tecnologici).
- **Importanza:** specifica l'importanza del requisito e può assumere i valori:
 - **O** - *Obbligatorio*, corrisponde ad un obiettivo e che deve essere rispettato per garantire il funzionamento minimo dell'applicazione;
 - **D** - *Desiderabile*, corrisponde ad un obiettivo rilevante per l'applicazione che però deve essere perseguito con priorità inferiore ai requisiti obbligatori;
 - **F** - *Facoltativo*, corrisponde ad un valore aggiunto per l'applicazione, ottenibile in rilasci futuri.
- **Codice:** rappresenta un codice numerico che identifica il requisito all'interno di una gerarchia. Esso è definito il modo tale per cui $RTIx.y$ sia un requisito che definisce con un grado maggiore di dettaglio alcuni aspetti del requisito $RTIx$.

3.2 Descrizione generale dei requisiti di Pollit

Come descritto in precedenza, **Pollit** si divide in due applicazioni. Pertanto il capitolo descrive le caratteristiche del prodotto separando la trattazione dei requisiti di *pollcenter* e *pollboy*.

I vincoli tecnologici individuati in accordo con l'azienda ospitante mirano alla fruibilità delle applicazioni in ambito desktop e mobile. Perciò il prototipo di **Pollit** dovrà essere realizzato con un layout responsive, che segua le indicazioni del Material Design di Google, e dovrà risultare funzionante sui moderni browser web. Molte delle funzionalità del prodotto sono riservate a rilasci futuri; l'obiettivo che azienda e studente hanno concordato è quello di arrivare ad ottenere il soddisfacimento dei requisiti obbligatori entro la fine dello stage.

Per capire il funzionamento degli applicativi e l'individuazione dei requisiti è bene introdurre la seguente terminologia:

- **poll**: il poll è un sondaggio a cui sottoporre un cliente. Ad un poll è possibile associare un nome ed una descrizione ed esso contiene una o più domande che verranno somministrate al cliente, secondo tre possibili tipologie:
 - *rating*, ovvero sarà possibile dare una valutazione entro un intervallo numerico a un quesito. Ad esempio “Quanto sei soddisfatto del servizio offerto? Rispondi con un valore da 0 a 5”;
 - *scelta testuale*, il cliente dovrà rispondere ad un quesito scegliendo una fra più alternative presentate in forma testuale. Ad esempio, “Preferisci il colore: A. Rosso, B. Verde”;
 - *scelta di immagini*, il cui funzionamento è analogo ad una scelta testuale, in questo caso l'utente dovrà scegliere una tra le immagini presentate.
- **kpi**: acronimo di *Key Performance Indicator*, sono indici che monitorano l'andamento di processi aziendali. Nel contesto di **Pollit**, si è voluto fornire uno strumento che consenta all'utente di creare dei kpi personali, che può associare a domande di sondaggi e di cui può monitorare le statistiche da una dashboard. Ad esempio, un utente può creare il kpi “Pulizia” ed associarlo alle domande di una campagna, come “Come valuti la pulizia della camera in cui hai soggiornato?” oppure “Hai trovato abbastanza puliti i bagni della piscina esterna?”.
- **campagna**: sarà possibile per l'utente creare una campagna promozionale che remunera l'utente in funzione delle risposte fornite ad un sondaggio. L'obiettivo è quello di acquisire ad esempio preferenze su prodotti o feedback sull'esperienza che il cliente ha avuto con l'azienda. Il premio che il cliente riceve verrà stabilito dall'azienda. Data la natura prototipale di **Pollit**, l'implementazione del sistema di remunerazione non verrà trattata.
- **URL**: ad ogni campagna sarà possibile associare un URL specifico. In questo modo i clienti potranno accedere ad un'istanza dell'applicazione *pollboy* tramite l'URL generato, che somministrerà sondaggi presenti nella campagna. Esiste una relazione univoca tra campagna e URL, infatti ad ogni URL potrà corrispondere una sola campagna;

3.3 Requisiti di pollcenter

pollcenter è un applicativo destinato al reparto marketing delle aziende che acquistano **Pollit**. Esso deve fornire la possibilità di creare dei questionari a cui il cliente finale potrà rispondere in cambio di una ricompensa alla fine delle domande (mediante l'applicazione *pollboy*). **Pollit** vuole essere uno strumento altamente configurabile e personalizzabile, quindi è possibile definire più poll in una campagna. Ad esempio, nel contesto della soddisfazione dei clienti di un albergo, un sondaggio può essere destinato alla pulizia dei locali mentre un altro alla qualità del ristorante. La *skip logic* del prodotto, ovvero la capacità di individuare la prossima domanda da somministrare al cliente in base alle sue scelte precedenti, può selezionare all'interno di una campagna domande provenienti da sondaggi distinti. Tuttavia non verrà trattata in questo documento, in quanto ancora in fase di sviluppo ed inerente al back-end dell'applicativo. Con *pollcenter*, quindi, è possibile creare campagne, sondaggi e monitorare mediante dashboard il loro andamento. Allo stato attuale non è ancora stato pienamente deciso quali statistiche mostrare, ma sono state fissate delle linee guida per i rilasci futuri:

- per campagne e poll si vogliono mostrare il numero di risposte che i clienti hanno dato a domande contenute in essi sia su base giornaliera che in totale;
- per i kpi si vogliono mostrare il numero di voti a domande che coinvolgono un kpi e un indicatore percentuale di soddisfazione, se il kpi è associato a domande di rating.

Una volta creata una campagna, sarà possibile associarla ad un URL univoco così da usufruire di *pollboy* e somministrare i quesiti ai clienti.

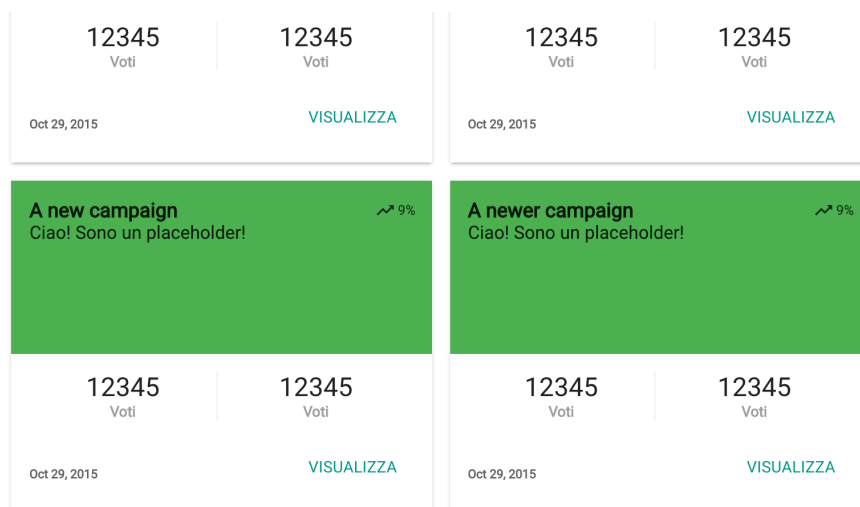


Figura 3.1: Esempio di dashboard che mostra le campagne create mediante card

3.3.1 Requisiti Funzionali

Id Requisito	Descrizione
RFO1	L'utente deve poter effettuare il login all'applicazione
RFO2	L'utente autenticato può visualizzare la lista delle campagne create
RFO2.1	L'utente autenticato può visualizzare le statistiche di una campagna dalla lista delle campagne
RFO3	L'utente autenticato può aggiungere una campagna
RFO4	L'utente autenticato può modificare una campagna
RFO4.1	L'utente autenticato può modificare il nome di una campagna
RFF5	L'utente autenticato può eliminare una campagna
RFO6	L'utente autenticato può visualizzare la lista dei kpi creati
RFO6.1	L'utente autenticato può visualizzare le statistiche di un kpi dalla lista dei kpi
RFO7	L'utente autenticato può aggiungere un kpi
RFO8	L'utente autenticato può modificare un kpi
RFO8.1	L'utente autenticato può modificare il nome di un kpi
RFO9	L'utente autenticato può eliminare un kpi
RFO10	L'utente autenticato può visualizzare la lista degli URL associati alle campagne
RFO11	L'utente autenticato può aggiungere un URL associato a una campagna
RFO11.1	L'utente autenticato può scegliere un nome per l'URL associato alla campagna
RFO11.2	L'utente autenticato può scegliere una campagna da associare all'URL
RFO12	L'utente autenticato modificare un URL associato a una campagna
RFO12.1	L'utente autenticato può modificare il nome di un URL associato a una campagna
RFD12.2	L'utente autenticato scegliere una campagna diversa da associare all'URL
RFO13	L'utente autenticato può visualizzare la lista dei poll di una campagna
RFO13.1	L'utente autenticato può visualizzare le statistiche di un poll dalla lista dei poll di una campagna
RFO14	L'utente autenticato può aggiungere un poll una campagna
RFO14.1	L'utente autenticato può inserire un nome per il poll
RFO15	L'utente autenticato può modificare un poll
RFO15.1	L'utente autenticato può aggiungere una domanda di tipo scelta multipla testuale
RFO15.2	L'utente autenticato può aggiungere una domanda di tipo scelta multipla con immagini
RFO15.3	L'utente autenticato può aggiungere una domanda di tipo rating
RFO15.4	L'utente autenticato può aggiungere una descrizione per il poll
RFO15.5	L'utente autenticato può modificare il nome del poll
RFO15.6	L'utente autenticato può eliminare una domanda
RFO16	L'utente autenticato può modificare una domanda

Id Requisito	Descrizione
RFO16.1	L'utente può modificare una domanda di tipo scelta multipla testuale
RFO16.1.1	L'utente autenticato può aggiungere una scelta testuale
RFO16.1.2	L'utente autenticato può eliminare una scelta testuale
RFO16.2	L'utente autenticato può modificare una domanda di tipo scelta multipla di immagine
RFO16.2.1	L'utente autenticato può aggiungere una scelta di immagine
RFO16.2.1.1	L'utente autenticato può effettuare l'upload di una immagine
RFO16.2.2	L'utente autenticato può eliminare una scelta di immagine
RFO16.3	L'utente autenticato può modificare il testo di una domanda
RFO16.4	L'utente autenticato può associare un kpi ad una domanda
RFO17	L'utente autenticato può eliminare un poll di una campagna

Tabella 3.1: Requisiti Funzionali di pollcenter

3.3.2 Requisiti di Vincolo

Id Requisito	Descrizione
RVO1	L'applicazione deve poter funzionare correttamente nel browser Google Chrome versione 45 o superiore
RVO2	L'applicazione deve poter funzionare correttamente nel browser Google Chrome per dispositivi mobili Android versione 4.4.2 o superiore
RVO3	L'applicazione deve poter funzionare correttamente nel browser Google Chrome per dispositivi mobili iOS versione 8 o superiore
RVF4	L'applicazione deve poter funzionare correttamente nel browser Google Chrome per dispositivi mobili Windows Phone versione 8 o superiore
RVO5	L'applicazione deve poter funzionare correttamente nel browser Safari versione 8 o superiore
RVD6	L'applicazione deve poter funzionare correttamente nel browser Safari per dispositivi mobili iOS versione 8 o superiore
RVF7	L'applicazione deve poter funzionare correttamente nel browser Internet Explorer versione 9 o superiore
RVF8	L'applicazione deve poter funzionare correttamente nel browser Internet Explorer per dispositivi mobili Windows Phone versione 8 o superiore
RVO9	L'interfaccia grafica dell'applicazione deve essere fluida e non bloccarsi
RVO10	L'interfaccia grafica dell'applicazione deve responsive e adattarsi a dispositivi desktop, a smartphone e a tablet
RVO11	L'interfaccia grafica dell'applicazione deve essere realizzata secondo i canoni del Material Design di Google

Tabella 3.2: Requisiti di Vincolo per pollcenter

3.3.3 Riepilogo requisiti

Sono stati individuati 53 requisiti. Essi sono stati ripartiti tra le tipologie precedentemente indicate secondo quanto riportato dalle seguenti tabelle:

Importanza	#
Obbligatori	47
Desiderabili	2
Facoltativi	4
Totale	53

Tabella 3.3: Numero di requisiti per importanza

Tipologia	#
Funzionali	42
Vincolo	11
Totale	53

Tabella 3.4: Numero di requisiti per tipologia

3.4 Requisiti di pollboy

pollboy è l'applicativo che verrà utilizzato dai clienti delle aziende che acquistano **Pollit**. Il cliente riceverà un URL dall'azienda dal quale avrà accesso ad un questionario facente parte di una campagna promozionale. Al termine di questo riceverà un premio stabilito dall'azienda (ad esempio un coupon).

pollboy è pensato per essere utilizzato prevalentemente su dispositivi mobili, poiché il cliente potrebbe ricevere l'invito a partecipare alla campagna mediante mezzi pubblicitari, come cartelloni posti in festival o fiere oppure giornali. L'interfaccia dell'applicazione deve quindi rendere accattivante e piacevole l'esperienza di partecipazione del cliente. Per fare questo, la votazione dovrà avvenire mediante l'uso di *gesture* comuni alle applicazioni per smartphone. Inoltre verrà sottoposto un solo quesito alla volta per limitare lo scroll della pagina e per consentire al back-end dell'applicazione di poter scegliere la domanda successiva esaminando le risposte precedenti. Nonostante l'inclinazione prettamente mobile di *pollboy*, le sue funzionalità lato desktop non devono essere in alcun modo pregiudicate.

Il login potrà essere effettuato mediante le *API* dei più famosi social network: in questo modo il back-end potrà avere accesso ad informazioni del profilo utente del cliente, fornendo metriche aggiuntive all'azienda e incrementare gli elementi per l'algoritmo di *skip logic*.

3.4.1 Requisiti Funzionali

Id Requisito	Descrizione
RFO1	L'utente deve poter effettuare il login
RFO1.1	L'utente deve poter effettuare il login tramite Facebook
RFF1.2	L'utente deve poter effettuare il login tramite i servizi Google
RFD1.3	L'utente deve poter effettuare il login tramite email
RFO2	L'utente autenticato deve poter partecipare ad un questionario
RFO3	L'utente autenticato deve poter rispondere a una domanda
RFO3.1	L'utente autenticato deve poter rispondere a una domanda di rating
RFO3.2	L'utente autenticato deve poter rispondere a una domanda a scelta multipla testuale
RFO3.3	L'utente autenticato deve poter rispondere a una domanda a scelta multipla con immagini
RFO4	L'utente autenticato deve poter portare a termine un questionario
RFD5	L'utente autenticato deve poter ricevere un premio al termine del questionario

Tabella 3.5: Requisiti Funzionali di pollboy

3.4.2 Requisiti di Vincolo

Id Requisito	Descrizione
RVO1	L'applicazione deve poter funzionare correttamente nel browser Google Chrome versione 45 o superiore
RVO2	L'applicazione deve poter funzionare correttamente nel browser Google Chrome per dispositivi mobili Android versione 4.4.2 o superiore
RVO3	L'applicazione deve poter funzionare correttamente nel browser Google Chrome per dispositivi mobili iOS versione 8 o superiore
RVF4	L'applicazione deve poter funzionare correttamente nel browser Google Chrome per dispositivi mobili Windows Phone versione 8 o superiore
RVO5	L'applicazione deve poter funzionare correttamente nel browser Safari versione 8 o superiore
RVD6	L'applicazione deve poter funzionare correttamente nel browser Safari per dispositivi mobili iOS versione 8 o superiore
RVF7	L'applicazione deve poter funzionare correttamente nel browser Internet Explorer versione 9 o superiore
RVF8	L'applicazione deve poter funzionare correttamente nel browser Internet Explorer per dispositivi mobili Windows Phone versione 8 o superiore
RVO9	L'interfaccia grafica dell'applicazione deve essere fluida e non bloccarsi
RVO10	L'interfaccia grafica dell'applicazione deve responsive e adattarsi a dispositivi desktop, a smartphone e a tablet
RVO11	L'interfaccia grafica dell'applicazione deve essere realizzata secondo i canoni del Material Design di Google

Tabella 3.6: Requisiti di Vincolo per pollboy

3.4.3 Riepilogo requisiti

Sono stati individuati 22 requisiti. Essi sono stati ripartiti tra le tipologie precedentemente indicate secondo quanto riportato dalle seguenti tabelle:

Importanza	#
Obbligatoria	16
Desiderabili	3
Facoltativi	3
Totale	22

Tabella 3.7: Numero di requisiti per importanza

Tipologia	#
Funzionali	11
Vincolo	11
Totale	22

Tabella 3.8: Numero di requisiti per tipologia

Capitolo 4

Progettazione

Il capitolo inizia introducendo delle considerazioni di cui azienda e studente hanno tenuto conto durante la progettazione, per poi descrivere l'architettura delle applicazioni. Verranno esposte sia le scelte per la progettazione dell'architettura e delle componenti, sia i mockup delle interfacce sui quali il team dedicato a **Pollit** si è basato per la sua realizzazione.

4.1 Considerazioni generali

Nella progettazione di *pollcenter* e *pollboy* si è cercato di ottenere un'architettura facilmente estendibile da incrementi di funzionalità futuri. Inoltre, si è cercato di mettere in pratica una serie di best-practices del framework AngularJS.

L'approccio alla progettazione per entrambe le applicazioni è basato sulla suddivisione dell'applicazione in moduli che rispecchino ognuno una funzionalità che l'applicazione deve presentare. La scelta è dovuta alla volontà di utilizzare il framework e il linguaggio al meglio delle loro possibilità, senza snaturare i comuni design pattern forzandone l'utilizzo e verrà trattata in modo approfondito nelle sezioni seguenti.

4.1.1 Suddivisione in moduli

Come già spiegato, l'applicazione verrà suddivisa in **moduli**, rappresentanti ognuno una funzionalità che l'applicazione dovrà avere. Ogni modulo conterrà la logica le componenti che realizzano la funzionalità ricercata, ovvero controller, service e directive. Esse potranno avere dipendenze entranti ed uscenti verso le componenti di altri moduli. È stato preferito questo approccio rispetto ad una struttura **MVC** poiché è una best-practice consigliata per il framework e per testarne effettivamente l'efficacia. Utilizzando un'architettura di tipo **MVC**, la struttura a cartelle dell'applicazione dovrebbe essere ramificata in controller, service, directive e template, ad esempio:

```
1 templates/  
2   _login.html  
3   _feed.html  
4 app/  
5   app.js  
6   controllers/  
7     LoginController.js  
8     FeedController.js  
9   directives/
```

```
10      FeedEntryDirective.js
11      services/
12          LoginService.js
13          FeedService.js
14      filters/
15          CapatalizeFilter.js
```

Codice 4.1: Esempio di suddivisione in directory per componenti di un'applicazione

Una suddivisione integralmente per moduli, si traduce invece nell'esempio:

```
1 app/
2   app.js
3   Feed/
4       _feed.html
5       FeedController.js
6       FeedEntryDirective.js
7       FeedService.js
8   Login/
9       _login.html
10      LoginController.js
11      LoginService.js
12   Shared/
13       CapatalizeFilter.js
```

Codice 4.2: Esempio di suddivisione in moduli di un'applicazione

e consente di ottenere i seguenti vantaggi:

- la struttura dell'applicazione è facilmente leggibile;
- risulta facile navigare tra i file e cercare quello che occorre modificare;
- tutti i file relativi ad una funzionalità risiederanno nella cartella del modulo corrispondente;
- è possibile distribuire i compiti di sviluppare funzionalità diverse a diversi membri del team;
- successivi incrementi di funzionalità dell'applicazione corrispondono all'aggiunta di moduli;

Come si può notare, la cartella del modulo *Feed* contiene controller, directive, service e template che verranno compilati in view relativi. In progetti complessi però i file di template potrebbero essere molti (considerando anche quelli utilizzati per la compilazione delle directive definite dall'utente programmatore), perciò si è deciso di ricavare una struttura di cartelle che separi le componenti dell'applicazione dai template delle view:

```
1 app/
2   js/
3       app.js
4       moduleA/
5           aController.js
6           aService.js
7           aDirective.js
8       moduleB/
9           bController.js
10          bService.js
```

```
11  
12     template/  
13         moduleA/  
14             templateViewA.html  
15             directives/  
16                 aDirectiveTpl.html  
17         moduleB/  
18             templateViewB.html  
19  
20     css/  
21         app.css  
22  
23     img/
```

Codice 4.3: Suddivisione in directory adottata

L'applicazione sarà quindi costituita da:

- **js**: questa è la cartella radice dei moduli dell'applicazione. Essa deve contenere il file principale **app.js**, che esplicherà le dipendenze verso i moduli, e i moduli. Ogni modulo potrà contenere i file dei propri controller, service e directive, esplicitando nel nome di ogni file la loro funzione utilizzando la notazione camel case:
 - *nomeDelServiceService.js*
 - *nomeDelControllerController.js*
 - *nomeDellaDirectiveDirective.js*
- **template**: questa cartella contiene i template HTML che verranno compilati da Angular in view dell'applicazione. I template verranno suddivisi in moduli ed eventualmente quelli relativi a directive verranno separati in apposite cartelle;
- **css**: in questa cartella sono contenuti i fogli di stile;
- **img**: le immagini utilizzate dai template delle view verranno riposte nella cartella specifica.

4.1.2 Utilizzo dei controller e dei service

Nel realizzare le applicazioni *pollboy* e *pollcenter*, i **service** avranno il compito di esporre un'interfaccia che consente di comunicare con le [API](#) fornite da Wannaup S.r.l.s., effettuando operazioni [CRUD](#) sui dati. I dati in questione, sono esposti in formato JSON.

I controller, invece, verranno utilizzati per compiere la *business logic* sul model dell'applicazione (le proprietà dello scope), parte dell'*application logic* assieme alle view e utilizzeranno i service per sfruttare i dati proveniente dalle [API](#).

Un semplice esempio di interazione tra le componenti è il seguente: supponiamo che si voglia ottenere e modificare la risorsa *A*. Il controller utilizza il service adeguato, effettuando una richiesta GET all'[API](#) che restituisce un oggetto JSON rappresentante la risorsa *A*. L'oggetto può essere aggiunto lo scope, così che esso possa essere modificato sfruttando il two-way data-binding. Se si desidera salvare in modo permanente le modifiche ad *A*, dovrà essere effettuata una richiesta PUT alle [API](#), mediante il service apposito.

4.2 API di Pollit

Di seguito vengono esposte le [API](#) utilizzate dallo studente durante lo sviluppo del prototipo delle applicazioni. Esse si rifanno ad un modello [REST-Like](#) e non [REST](#) puro in quanto tutte le richieste HTTP vengono firmate mediante la tecnica **JWT**, presentata in precedenza ([JWT Authentication](#)).

Grazie al token JWT non è necessario trasmettere informazioni riguardo l'utente nelle richieste alle [API](#) in quanto già presenti in forma criptata nell'header HTTP delle richieste.

Le [API](#) verranno riportate nella forma `metodo_HTTP url/api`.

4.2.1 Campagne

Le [API](#) che consentono di interagire con le campagne sono le seguenti:

- **GET `api/campaigns`**
Restituisce un array di oggetti JSON che descrivono le campagne dell'utente che ha effettuato il login, con il seguente schema:

```
1  [  
2  {  
3    "id": string,  
4    "name": string,  
5    "status": boolean,  
6    "started": string,  
7    "stopped": string  
8  }  
9  ]  
10
```

Codice 4.4: JSON schema dell'API GET `api/campaigns`

dove:

- *id*: è l'identificativo della campagna nel database;
 - *name*: è il nome della campagna;
 - *status*: indica se la campagna è aperta o chiusa;
 - *started*: è il timestamp della data in cui la campagna è iniziata;
 - *stopped*: è il timestamp della data in cui la campagna è iniziata.
- **GET `api/campaigns/{campaignId}`**
Restituisce un oggetto JSON che descrive la campagna con identificativo *campaignId*:

```
1  {  
2    "id": string,  
3    "name": string,  
4    "status": boolean,  
5    "started": string,  
6    "stopped": string  
7  }  
8
```

Codice 4.5: JSON schema dell'API GET `api/campaigns/campaignId`

dove:

- *id*: è l'identificativo della campagna nel database;
- *name*: è il nome della campagna;
- *status*: indica se la campagna è aperta o chiusa;
- *started*: è il timestamp della data in cui la campagna è iniziata;
- *stopped*: è il timestamp della data in cui la campagna è iniziata.

- **PUT `api/campaigns/{campaignId}`**

Consente di aggiornare la campagna con identificativo *campaignId*. L'oggetto JSON da inviare nella richiesta deve avere la stessa forma dell'oggetto restituito per le campagne, con i campi aggiornati.

- **POST `api/campaigns`**

Consente di creare una nuova campagna . L'oggetto JSON da inviare nella richiesta deve avere la forma:

```
1 {  
2   "name": string,  
3 }  
4
```

Codice 4.6: JSON schema dell'API POST `api/campaigns/`

dove:

- *name*: è il nome della campagna.

- **GET `api/campaigns/{campaignId}/stats/votesvertime`**

Consente di recuperare il numero di voti complessivi ai poll della campagna *campaignId* nel tempo. Restituisce un array di oggetti JSON nella forma:

```
1 [  
2 {  
3   "ts": string,  
4   "n_votes": number  
5 }  
6 ]  
7
```

Codice 4.7: JSON schema dell'API GET `api/campaigns/campaignId/stats/votesvertime`

dove:

- *ts*: è il timestamp della data di cui si ottiene il numero di voti;
- *n_votes*: è un intero che indica il numero complessivo di voti per una data.

- **GET `api/campaigns/{urlH}`**

Viene utilizzata da *pollboy*, dato un URL *urlH* associato ad una campagna, restituisce una domanda di un poll della campagna associata. Viene restituito un oggetto JSON nella forma:

```

1  [
2  {
3      "qId": string,
4      "qText": string,
5      "choices": [{"id": string, "value": string}]
6  }
7  ]
8

```

Codice 4.8: JSON schema dell'API GET `api/campaigns/urlH`

dove:

- *qId*: è l'identificativo della domanda nel database;
- *qText*: è il testo della domanda;
- *choices*: è un array di oggetti che rappresentano le possibili risposte. L'array è vuoto nel caso di domande di tipo rating.

- **POST `api/campaigns/{campaignId}/vote`**

Viene utilizzata da *pollboy*, per confermare la risposta di un utente ad una domanda. Viene inviato un oggetto JSON nella forma:

```

1  {
2      "qId": string,
3      "cId": string,
4  }
5
6  oppure in caso di domanda di rating
7
8  {
9      "qId": string,
10     "value": number,
11 }
12
13

```

Codice 4.9: JSON schema dell'API post `api/campaigns/campaignId/vote`

dove:

- *qId*: è l'identificativo della domanda nel database;
- *cId*: è l'identificativo della scelta dell'utente;
- *value*: è il valore di rating da 0 a 5 che l'utente ha espresso per la domanda.

4.2.2 Poll

Le [API](#) che consentono di interagire con i poll sono:

- **GET `api/campaigns/{campaignId}/polls`**

Restituisce un array di oggetti JSON che descrivono i poll della campagna *campaignId*, con il seguente schema:

```

1  [
2  {
3      "id": string,

```

```

4      "campaign_id":string,
5      "name":string,
6      "desc":string,
7      "last_voted":string,
8      "quests":[
9          {
10             "id": string,
11             "type": number,
12             "text": string,
13             "choices":[
14                 {
15                     "id": string,
16                     "text": string,
17                     "img": string
18                 }
19             ],
20             "kpis": [
21                 {
22                     "id": string
23                     "name": string
24                 }
25             ]
26         }
27     ]
28 }
29 ]
30

```

Codice 4.10: JSON schema dell'API GET `api/campaigns/campaignId/polls`

dove:

- *id*: è l'identificativo del poll nel database;
- *campaign_id*: è l'identificativo della campagna in cui il poll è stato creato nel database;
- *name*: è il nome del poll;
- *desc*: rappresenta una breve descrizione del poll;
- *last_voted*: è il timestamp della data in cui la campagna è avvenuta l'ultima votazione;
- *quests*: array di oggetti JSON che rappresentano le domande di un poll. Ogni domanda avrà:
 - * *id*: l'identificativo della domanda;
 - * *type*: un intero da 0 a 2, identifica il tipo di domanda. 0 per le domande a scelta testuale, 1 per le domande a scelta di immagini e 2 per le domande di rating;
 - * *choices*: è un array di oggetti JSON che rappresentano le scelte con cui si può rispondere a una domanda. Ogni scelta avrà un identificativo, un testo, nel caso di scelta testuale o un'immagine, nel caso di scelta di immagini;
 - * *kpis*: un array di oggetti JSON che indicano quali sono i kpi associati alla domanda.
- **POST `api/campaigns/{campaignId}/polls`** Consente di creare un nuovo poll nella campagna *campaignId*. L'oggetto JSON da inviare nella richiesta deve avere la forma:

```

1  {
2    "name": string,
3  }
4

```

Codice 4.11: JSON schema dell'API POST `api/campaigns/campaignId/polls`

dove:

– *name*: è il nome del poll.

- **GET `/api/campaigns/{campaignId}/polls/{pollId}`**
Restituisce un oggetto JSON che descrive il poll *pollId* della campagna con identificativo *campaignId*. Lo schema dell'oggetto è analogo a quello presentato nell'[API GET `api/campaigns/{campaignId}/polls`](#).
- **PUT `/api/campaigns/{campaignId}/polls/{pollId}`**
Consente di aggiornare e modificare il poll *pollId* della campagna con identificativo *campaignId*. L'oggetto JSON da inviare nella richiesta deve avere la stessa forma dell'oggetto restituito per i poll, con i campi aggiornati.
- **GET `/api/campaigns/{campaignId}/polls/{pollId}/stats/votesvertime`**
Consente di recuperare il numero di voti complessivi del poll *pollId* della campagna *campaignId* nel tempo. Restituisce un array di oggetti JSON nella forma:

```

1  [
2    {
3      "ts": string,
4      "n_votes": number
5    }
6  ]
7

```

Codice 4.12: JSON schema dell'API GET `api/campaigns/campaignId/polls/pollId/stats/votesvertime`

dove:

- *ts*: è il timestamp della data di cui si ottiene il numero di voti;
- *n_votes*: è un intero che indica il numero complessivo di voti per una data.

4.2.3 Kpi

Di seguito vengono elencate le [API](#) che consentono di interagire con i kpi:

- **GET `api/kpis`**
Restituisce un array di oggetti JSON che descrivono i kpi dell'utente che ha effettuato il login, con il seguente schema:

```

1  [
2    {
3      "id": string,
4      "name": string,
5    }
6  ]

```


7

Codice 4.13: JSON schema dell'API GET `api/kpis`

dove:

- *id*: è l'identificativo del kpi nel database;
- *name*: è il nome del kpi.

- **GET `api/kpis/{kpiId}`**

Restituisce un oggetto JSON che descrive il kpi con identificativo *kpiId*. Lo schema dell'oggetto è analogo a quello visto per l'API GET `api/kpis`.

- **POST `api/campaigns`**

Consente di creare un nuovo kpi . L'oggetto JSON da inviare nella richiesta deve avere la forma:

```
1 {  
2   "name": string,  
3 }  
4
```

Codice 4.14: JSON schema dell'API POST `api/kpis/`

dove:

- *name*: è il nome del kpi.

4.2.4 URL

Le seguenti API che consentono di creare, modificare e recuperare gli URL di un utente:

- **GET `api/urlhandles`**

Restituisce un array di oggetti JSON che descrivono gli URL che l'utente ha associato a delle campagne, con il seguente schema:

```
1 [  
2 {  
3   "id": string,  
4   "url": string,  
5   "bound_campaign_id": string  
6 }  
7 ]  
8
```

Codice 4.15: JSON schema dell'API GET `api/kpis`

dove:

- *id*: è l'identificativo dell'URL nel database;
- *url*: è il valore testuale scelto per completare e distribuire l'URL;
- *bound_campaign_id*: è l'identificativo della campagna alla quale l'URL è associato.

- **POST `api/urlhandles`**

Consente di creare un nuovo URL . L'oggetto JSON da inviare nella richiesta deve avere la forma:

```
1 {  
2   "url": string,  
3   "bound_campaign_id": string  
4 }  
5
```

Codice 4.16: JSON schema dell'API POST `api/urlhandles/`

dove:

- *url*: è il valore testuale scelto per completare e distribuire l'URL;
 - *bound_campaign_id*: è l'identificativo della campagna alla quale si desidera associare l'URL.
- **GET `api/urlhandles/{urlId}`**
Restituisce un oggetto JSON che descrive l'URL con identificativo *urlId*. Lo schema dell'oggetto è analogo a quello visto per l'API GET `api/urlhandles`.
 - **PUT `api/urlhandles/{urlId}`**
Consente di aggiornare e modificare l'URL *urlId*. L'oggetto JSON da inviare nella richiesta deve avere la stessa forma dell'oggetto restituito per gli URL, con i campi aggiornati.
 - **DELETE `api/urlhandles/{urlId}`**
Consente di eliminare l'URL *urlId*.

4.3 pollcenter

pollcenter viene intesa come una dashboard e un insieme di strumenti per controllare le campagne, i poll in esse, i kpi e gli URL di un utente. Pertanto è stato immaginato un layout che mediante le card del Material Design potesse mostrare in semplici click un cruscotto su cui leggere statistiche e andamenti mediante grafici.

Poiché il progetto prevede il completamento di un prototipo, durante il suo sviluppo le uniche metriche prese in considerazione per la creazione della dashboard e dei grafici sono state il variare dei voti relativi a un poll nel tempo ed il variare dei voti complessivi dei poll di una campagna nel tempo. La priorità è stata data alla creazione di un'interfaccia pulita e alla realizzazione di un'applicazione scalabile in grado di adattarsi ad esporre grafici di più metriche senza cambiamenti radicali, bensì integrando semplicemente nuove funzionalità.

Per l'applicazione *pollcenter* sono stati individuati i seguenti moduli:

- *login*: il modulo deve realizzare le funzionalità di login e logout dell'utente;
- *campaigns*: il modulo deve consentire di creare, modificare e visualizzare le campagne di un utente e le statistiche correlate;
- *kpis*: il modulo deve consentire di creare, modificare e visualizzare i kpi di un utente e le statistiche correlate ad essi;

- *urls*: il modulo deve consentire di creare, visualizzare ed eliminare gli URL che un utente può associare alle proprie campagne;
- *polls*: il modulo deve consentire di creare, visualizzare ed eliminare i poll di una campagna, visualizzarne le statistiche. Inoltre deve consentire di poter creare per ogni poll, domande dei tipi individuati in [Descrizione generale dei requisiti di Pollit](#);
- *common*: il modulo contiene le componenti di utilizzo comune tra i moduli, ad esempio directive che realizzano particolari componenti grafici.

Di conseguenza, la struttura dell'applicazione risultante è:

```
1 app/  
2   js/  
3     app.js  
4     login/  
5       loginService.js  
6       loginController.js  
7     campaigns/  
8       campaignsListControllerr.js  
9       editCampaignController.js  
10      campaignsService.js  
11     common/  
12       statsCardDirective.js  
13   kpis/  
14     kpisListController.js  
15     kpisService.js  
16   polls/  
17     editPollController.js  
18     pollsService.js  
19     questEditorDirective.js  
20     textChoiceDirective.js  
21     imageChoiceDirective.js  
22     kpiTagsDirective.js  
23   urls/  
24     urlsListController.js  
25     urlsService.js  
26  
27   template/  
28     login/  
29       loginView.html  
30     campaigns/  
31       campaigns.html  
32       editCampaign.html  
33     common/  
34       directives/  
35         statsCardDirective.html  
36     kpis/  
37       kpis.html  
38     polls/  
39       editPoll.html  
40       directives/  
41         questEditorDirective.html  
42         textChoiceDirective.html  
43         imageChoiceDirective.html  
44         kpiTagsDirective.html  
45     urls/  
46       urls.html  
47  
48   css/
```

```
49 | app.css
```

Codice 4.17: Struttura di pollcenter

In seguito verranno mostrati in dettaglio i moduli e i loro service, controller e i template che realizzeranno le view.

4.3.1 Modulo login

Questo modulo realizza il login all'applicazione:

- **service**
 - *loginService*: il service effettua richieste HTTP di tipo POST e con un corpo contenente i campi dati *username* e *password* dell'utente, codificati come form-data.
Nel caso che l'autenticazione sia andata a buon fine, il server risponde con un header HTTP 200, mentre se l'autenticazione fallisce risponde con un header 401.
- **controller**
 - *loginController*: utilizza *loginService* e se l'autenticazione va a buon fine, reindirizza alla home page dell'applicazione.
- **template**
 - *loginView*: è la pagina di login dell'applicazione, verrà compilata in una view su cui opera *loginController*.

**Figura 4.1:** Layout della pagina di login a *pollcenter*

4.3.2 Modulo common

Il modulo common contiene tutto il codice che può essere condiviso tra i moduli dell'applicazione. In questo modo può avvenire il riuso di codice e si possono diminuire eventuali dipendenze concentrando in un unico modulo. In particolare, *common* conterrà le directive utilizzate da view di moduli diversi.

Per rappresentare le liste delle campagne disponibili, dei poll in una campagna e dei kpi di un utente, è stata valutata la possibilità di creare una directive comune che renderizzi una card contenente le informazioni e le statistiche da mostrare all'utente, per comporre una funzionale dashboard.

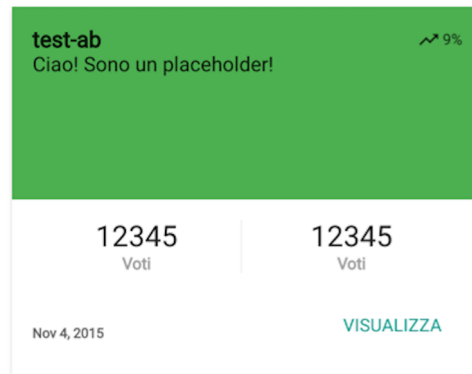


Figura 4.2: Card che espone le statistiche di campagne, poll e kpi

In questo modulo sono presenti:

- **directive**
 - *statsCardDirective*: directive che può essere utilizzata dalla view in modo configurabile. Realizza una card contenente le informazioni e le statistiche con cui la si configurerà.
- **template**
 - *statsCardDirective.html*: il template con cui verrà renderizzata la directive *statsCardDirective* all'interno delle view.

4.3.3 Modulo campaigns

Questo modulo deve gestire tutte le operazioni che un utente può compiere sulle proprie campagne, ovvero visualizzarle, crearle, modificarle ed eliminarle. Inoltre deve poter consentire di aggiungere dei poll a delle campagne. La scelta di aggiungere poll dalla lista delle campagne è dettata dal layout che si vuole ottenere, che deve rendere semplice ed intuitivo all'utente dell'applicazione.

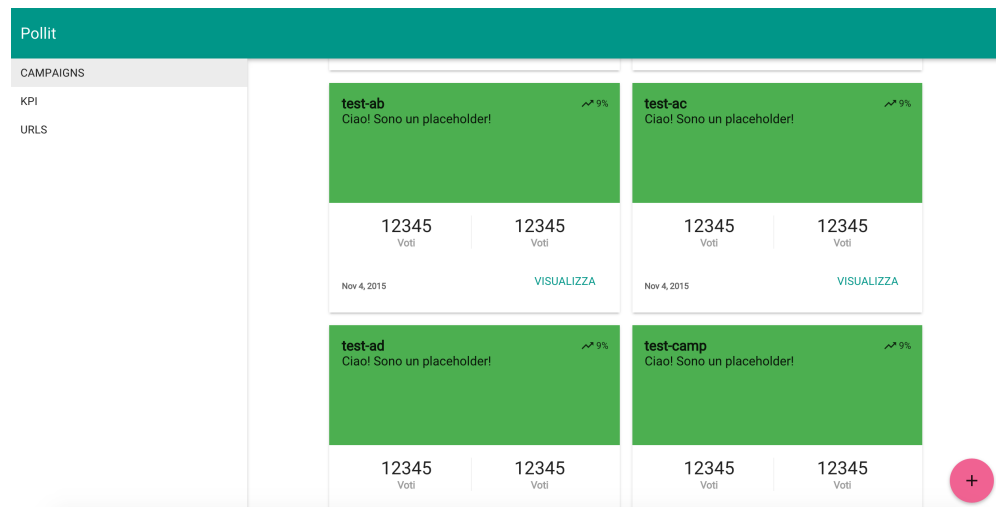


Figura 4.3: Dashboard in cui ogni card rappresenta una campagna. Utilizzando il FAB si possono aggiungere campagne

Come si può notare dal prototipo di interfaccia, l'utente che accederà all'applicazione troverà come pagina iniziale una dashboard con card che rappresentano le campagne e che esporranno per esse un grafico mostrante il variare del numero di voti alle domande dei poll nel tempo.

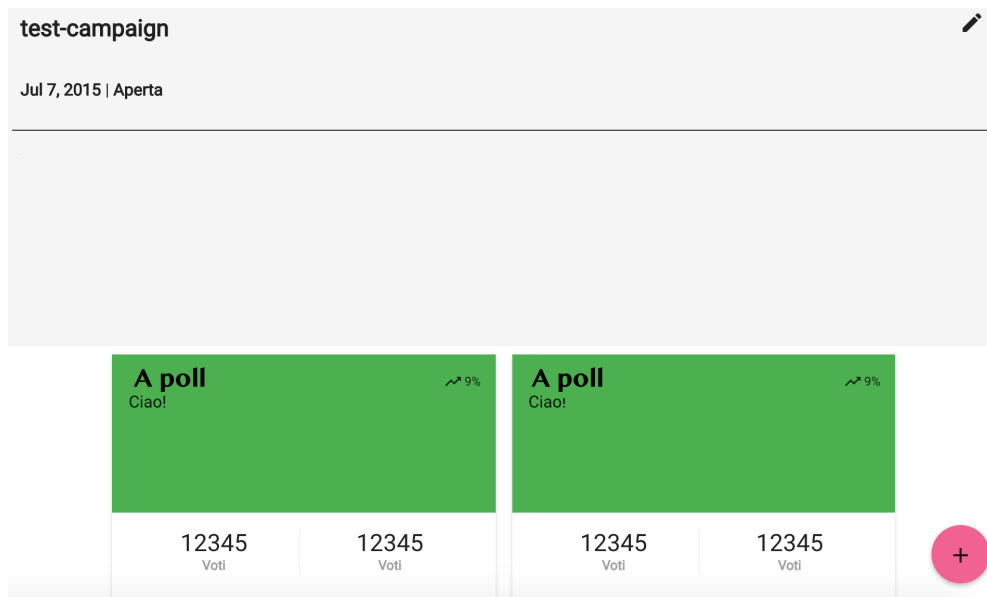


Figura 4.4: Una campagna nel dettaglio con i poll ad essa associati rappresentati dalle card

Le card consentiranno all'utente di accedere alla pagina di dettaglio di una campagna, dalla quale si potrà modificarne il nome, visualizzarne la lista dei poll ed aggiungerne di nuovi mediante il FAB.

In questo modulo sono presenti:

- **service**

- *campaignsService*: il service dovrà utilizzare le [API](#) e fornire un'interfaccia ai controller che consenta di ottenere la lista delle campagne dell'utente e di modificare e creare campagne;

- **controller**

- *campaignsListController*: questo controller deve occuparsi di recuperare mediante *campaignService* la lista delle campagne di un utente ed esporle mediante card del Material Design. Deve inoltre fornire la possibilità di crearne di nuove;
- *editCampaignController*: il controller ha lo scopo di modificare le informazioni di una campagna. Esso recupera mediante *campaignService* l'oggetto JSON che rappresenta la campagna e viene collegato alla view generata dal template *editCampaign.html*. Inoltre utilizzerà i service del modulo dei poll per ottenere la lista dei poll associati ad una campagna e per poterne aggiungere di nuovi. Esso deve quindi permettere di creare poll vuoti. La loro modifica sarà gestita dal modulo dedicato ai poll;

- **template**

- *campaigns.html*: template che verrà compilato nella view che rappresenta la dashboard delle campagne;
- *editCampaign.html*: template che verrà compilato nella view dalla quale è possibile modificare le informazioni di una campagna.

4.3.4 Modulo polls

Il modulo dedicato ai poll si occuperà di compiere le funzionalità inerenti alla creazione, modifica, eliminazione dei poll di una campagna. Esso utilizzerà le [API](#) per recuperare informazioni riguardo alla lista di poll di una campagna o ad un singolo poll.

Ogni questionario potrà contenere più domande, che verranno somministrate con *pollboy*. Esse saranno di tre tipi: a scelta testuale, a scelta di immagini o rating, come descritto in [Descrizione generale dei requisiti di Pollit](#).

Ad ogni domanda sarà possibile aggiungere uno o più kpi, che acquisteranno quindi peso ogni volta che a quella domanda sarà data una risposta. È stato scelto di rappresentare graficamente l'aggiunta di un kpi attraverso le *chips* o *tag* del Material Design, come mostra l'immagine seguente:

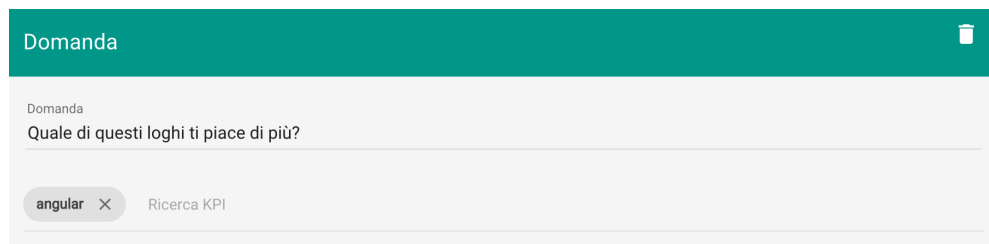


Figura 4.5: Esempio grafico di una domanda, con inserimento del titolo e aggiunta dei tag che rappresentano i kpi mediante autocompletamento

L'utente potrà cercare tra i kpi disponibili e aggiungerne ed eliminarne in modo dinamico mediante le chips.

Grazie ad AngularJS sarà possibile utilizzare un'unica view per la creazione e la modifica di un poll: i dati esistenti verranno acquisiti dalle [API](#), presentati e modificati attraverso le directive salvati sempre attraverso l'interfaccia delle [API](#).

In questo modulo sono presenti:

- **service**

- *pollsService*: il service dovrà utilizzare le [API](#) e fornire un'interfaccia ai controller che consenta di ottenere la lista dei poll di una campagna e di modificare, eliminare e creare un poll. Inoltre dovrà permettere di creare e recuperare le domande di un poll;

- **controller**

- *editPollController*: il controller ha lo scopo di modificare le informazioni di un poll. Esso recupera mediante *pollsService* l'oggetto JSON che rappresenta il poll e viene collegato alla view generata dal template *editPoll.html*. In questo modo verrà sfruttato il two-way data binding per modificare l'oggetto JSON. In particolare potrà essere fornito un nome per il poll, una descrizione e sarà possibile aggiungere ed eliminare domande mediante la directive *editQuestDirective*;

- **directive**

- *editQuestDirective*: questa directive consente la creazione di domande in un poll. Sarà possibile fornire un testo per la domanda e aggiungere kpi. Per

fare questo, la directive utilizzerà la directive *kpiTagsDirective*. La directive inoltre dovrà poter essere istanziata in modo da poter creare domande di rating e utilizzerà *textChoiceDirective* e *imageChoiceDirective* per la creazione di domande di tipo scelta testuale e scelta con immagini;

- *kpiTagsDirective*: questa directive utilizza i service del modulo dei kpi ([Modulo kpis](#)) per ottenere la lista dei kpi disponibili e crea un campo testuale ad autocompletamento che consente di cercare e aggiungere kpi ad una domanda, mediante chips del Material Design;
- *textChoiceDirective*: la directive consente di aggiungere ed eliminare scelte testuali per una domanda di tipo scelta testuale;
- *imageChoiceDirective*: la directive consente di aggiungere ed eliminare scelte di immagini per una domanda di tipo scelta di immagini. Per fare ciò, dovrà permettere all'utente di effettuare upload di immagini;

- **template**

- *kpiTagsDirective.html*: template della directive *kpiTagsDirective*;
- *questEditorDirective.html*: template della directive *editQuestDirective*;
- *textChoiceDirective.html*: template della directive *textChoiceDirective*;
- *imageChoiceDirective.html*: template della directive *imageChoiceDirective*.

4.3.5 Modulo kpis

Questo modulo dovrà implementare le funzionalità legate all'aggiunta, la modifica e la visualizzazione della dashboard dei kpi dell'utente che ha effettuato il login. La sua struttura e l'interfaccia utente ricalcheranno la dashboard delle campagne.

Sono stati individuati i componenti:

- **service**

- *kpisService*: il service dovrà utilizzare le [API](#) e fornire un'interfaccia ai controller che consenta di ottenere la lista dei kpi creati da un utente e di modificare, eliminare e creare kpi;

- **controller**

- *kpisListController*: questo controller deve occuparsi di recuperare mediante *kpisService* la lista dei kpi di un utente ed esporli mediante card del Material Design. Deve inoltre fornire la possibilità di crearne di nuovi o di eliminarne;

- **template**

- *kpis.html*: template che verrà compilato nella view che rappresenta la dashboard dei kpi.

4.3.6 Modulo urls

L'utente avrà la possibilità di creare degli URL per le proprie campagne, scegliendo un nominativo. In questo modo potrà distribuirli ai consumatori per sottoporli ai questionari con *pollboy*. Il vincolo che viene posto è l'univocità tra un URL e una

campagna.

Un esempio di utilizzo potrebbe essere il seguente: la compagnia “Summertime” crea una campagna denominata “Estate 2015” e dei poll associati ad essa. Il settore marketing sceglie il nome “summertime-estate” per l’URL ed associarvi la campagna, quindi distribuirà l’URL `pollit.fake/summertime-estate`.

Il modulo *urls* dovrà rendere disponibili le funzionalità per creare, modificare ed eliminare URL associati alle campagne dell’utente che ha effettuato il login e dovrà visualizzarli in forma di lista.


Url che hai creato		
frameworks	campagna collegata: 559be0acfec02512d7000002	



Figura 4.6: Esempio grafico di un URL per una campagna. Attraverso il FAB sarà possibile aggiungerne di nuovi

I componenti individuati sono:

- **service**

- *urlsService*: il service dovrà utilizzare le [API](#) e fornire un’interfaccia ai controller che consenta di ottenere la lista degli URL creati da un utente e di modificare, eliminare e creare URL;

- **controller**

- *urlsListController*: questo controller deve occuparsi di recuperare mediante *urlsService* la lista degli URL di un utente ed esporli mediante una lista in stile Material Design. Deve inoltre fornire la possibilità di crearne di nuovi o di eliminarne;

- **template**

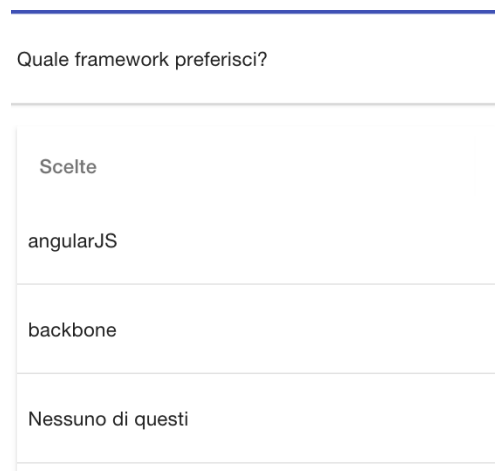
- *kpis.html*: template che verrà compilato nella view che rappresenta la lista degli URL.

4.4 pollboy

pollboy è l'applicazione che consente all'utente consumatore di partecipare ad un questionario e a cui accederà attraverso l'URL di una campagna. Dopo un login attraverso i più comuni social media, gli verrà somministrata una domanda alla volta. Al completamento per l'azienda sarà possibile fornire una ricompensa.

Poiché **Pollit** è ancora un prototipo, la progettazione e la realizzazione hanno tenuto conto di un unico social media per il login, ovvero *Facebook*.

Il questionario verrà presentato con un'interfaccia in Material Design ed è stata data molta enfasi all'esperienza utente su dispositivi mobili, principale ambiente di esecuzione per *pollboy*: sarà possibile rispondere alle domande da smartphone utilizzando le [gesture](#) a cui si è abituati utilizzando le comuni applicazioni per smartphone, ad esempio il [tap](#).



Quale framework preferisci?

Scelte
angularJS
backbone
Nessuno di questi

Figura 4.7: Esempio grafico dell'interfaccia di pollboy su smartphone. Sarà possibile esprimere la preferenza mediante gesture

I moduli individuati per *pollboy* sono:

- *login*: il modulo deve realizzare le funzionalità di login e logout dell'utente mediante le [API](#) fornite da Facebook;
- *campaigns*: il modulo deve individuare la campagna promozionale a cui l'utente sta partecipando, consentire di recuperare le domande a cui l'utente dovrà rispondere o votare e confermare la votazione;
- *polls*: il modulo conterrà directive e template che rappresenteranno graficamente le domande;

Quindi è stata individuata la seguente struttura per l'applicazione:

```
1 app/  
2   js/  
3     app.js  
4     login/  
5       loginService.js  
6       loginController.js  
7     campaigns/
```

```
8      campaignController.js
9      campaignService.js
10     polls/
11         pollQuestionDirective.js
12
13     template/
14         login/
15             loginView.html
16         campaigns/
17             campaign.html
18             campaignComplete.html
19         polls/
20             directives/
21                 pollQuestionDirective.html
22
23     css/
24         app.css
```

Codice 4.18: Struttura di pollboy

4.4.1 Modulo login

Questo modulo realizza il login all'applicazione:

- **service**
 - *loginService*: il service effettua richieste alle [API](#) di Facebook con cui l'utente potrà effettuare il login;
- **controller**
 - *loginController*: utilizza *loginService* e se l'autenticazione va a buon fine, reindirizza alla home page dell'applicazione.
- **template**
 - *loginView*: è la pagina di login dell'applicazione, verrà compilata in una view su cui opera *loginController*.

4.4.2 Modulo campaigns

Questo modulo si occupa di recuperare le domande a cui il cliente consumatore dovrà rispondere, mediante le [API](#) fornite, nel contesto della campagna relativa all'URL utilizzato. Una sola domanda verrà somministrata per volta. Al termine del questionario l'utente verrà reindirizzato alla pagina del premio ottenuto.

I componenti individuati sono

- **service**
 - *campaignsService*: il service recupera le domande da somministrare mediante [API](#) e conferma la votazione dell'utente, consentendo il passaggio alla domanda successiva o la terminazione del questionario. Esso deve esporre un'interfaccia per i controller;
- **controller**

- *campaignsController*: utilizza *campaignsService* per ottenere le domande da sottoporre

- **template**

- *campaign.html*: è la pagina che visualizza le domande e utilizzerà le directive del modulo polls per rappresentarle. Verrà compilata in una view su cui opera *campaignsController*;
- *campaignComplete.html*: è la pagina che mostra il premio ricevuto al termine del questionario. Essendo l'applicazione un prototipo, essa sarà vuota.

4.4.3 Modulo polls

Questo modulo conterrà le directive che rappresentano graficamente in Material Design le domande e le scelte di risposta.

I componenti individuati sono:

- **directive**

- *pollQuestionDirective*: questa directive crea una lista in Material Design che consente di scegliere una tra più alternative, in caso di scelta testuale o di immagini, o di valutare con un punteggio da 0 a 5 una domanda di rating. Per un utilizzo di *pollboy* mediante desktop, la directive dovrà consentire la votazione mediante click del mouse, mentre su smartphone e tablet dovrà essere garantito il funzionamento tramite [gesture tap](#)

- **template**

- *pollQuestionDirective.html*: template per la directive *pollQuestionDirective*.

Capitolo 5

Realizzazione

Il capitolo descrive le attività svolte dallo studente e le principali difficoltà incontrate durante lo sviluppo delle applicazioni. Per prima verrà trattata *pollcenter*, in seguito *pollboy* e saranno disponibili gli screenshot dei risultati finali ottenuti.

La codifica è avvenuta per moduli: prima di passare al modulo successivo il risultato è stato visionato dall'azienda e dal tutor che hanno effettuato i test necessari ed eventualmente proposto modifiche.

Saranno riportati solamente gli esempi di codice ritenuti più significativi.

5.1 pollcenter

5.1.1 Realizzazione dell'autenticazione alle API con JWT

Prima di iniziare lo sviluppo effettivo del prototipo *pollcenter*, è stato necessario realizzare il sistema di autenticazione per le [API](#) lato front-end. La tecnica scelta è stata l'autenticazione JWT. AngularJS non fornisce nativamente un modulo che consente di implementarla, per questo motivo è stato impiegato il modulo **angular-jwt**.

Una volta effettuato il login dell'applicazione, il token viene salvato nel *local storage* del browser da cui essa viene eseguita. Quando i service effettueranno chiamate alle [API](#) di Wannaup S.r.l.s., interverrà un componente detto *interceptor* che preleverà il token e lo aggiungerà nell'header della richiesta HTTP. L'*interceptor* di AngularJS viene configurato nel file principale dell'applicazione **app.js** nel modo seguente:

```
1 .config(['$httpProvider', 'jwtInterceptorProvider', function($httpProvider,
2   jwtInterceptorProvider){
3   jwtInterceptorProvider.tokenGetter = ['config', '$location', function(
4     config, $location) {
5     if (config.withCredentials !== undefined && config.withCredentials ===
6       false)
7       return null;
8     if (config.url.substr(config.url.length - 5) == '.html') {
9       return null;
10    }
11    return localStorage.getItem('auth_token');
12  }];
13  $httpProvider.interceptors.push('jwtInterceptor');
14 }])
```

Codice 5.1: Realizzazione dell'autenticazione JWT

\$httpProvider è un modulo nativo di Angular che consente di effettuare richieste HTTP. In particolare esso mantiene un array di interceptor, che possono effettuare operazioni quando avviene una effettiva chiamata ad **API**. **jwtInterceptorProvider** configura un interceptor in grado di inserire il token JWT all'interno delle richieste HTTP, che viene poi aggiunto a quelli di **\$httpProvider**.

Tuttavia, possedere un token non garantisce in tutti i casi l'accesso alle **API**. Ad esempio, la validità temporale del token potrebbe essere scaduta. In tal caso si rende necessario un interceptor che alla risposta con codice di errore HTTP 401 (mancanza di autorizzazione) dell' **API**, reindirizzi l'utente alla pagina di login per rinnovare il token.

```

1  .factory('authHttpResponseInterceptor', ['$q', '$window', function($q, $window) {
2      return {
3          response: function(response) {
4              if (response.status === 401) {
5                  console.log("Response 401");
6              }
7              return response || $q.when(response);
8          },
9          responseError: function(rejection) {
10             if (rejection.status === 401) {
11                 console.log("Response Error 401", rejection);
12                 $window.location.href = '/login';
13             }
14             return $q.reject(rejection);
15         }
16     };
17 })
18
19 .....
20
21 $httpProvider.interceptors.push('authHttpResponseInterceptor');
```

Codice 5.2: Realizzazione dell'autenticazione JWT con interceptor per errore 401

5.1.2 Routing dell'applicazione

Successivamente è stato necessario implementare il sistema di routing dell'applicazione. Per routing si intende l'associare un particolare URL ad una view in modo che, al cambiamento dell'URL nella barra degli indirizzi, l'applicazione possa reindirizzare l'utente alla pagina corretta.

Poiché il routing nativo di AngularJS non è stato ritenuto sufficientemente prestante, si è optato per l'utilizzo del modulo di terze parti **ui-router**. Esso fornisce un service **\$stateProvider** che consente di creare degli "stati" dell'applicazione.

```

1  .config(['$stateProvider', '$httpProvider', 'jwtInterceptorProvider',
2      function($stateProvider, $httpProvider, jwtInterceptorProvider){
3      $stateProvider
4          .state('polls', {
5              url: "/campaigns/:campaignId/polls",
6              templateUrl: "/assets/tpl/polls/polls.html"
7          })
8          .state('viewPoll', {
```



```

8      url: "/campaigns/:campaignId/polls/:pollId",
9      templateUrl: "/assets/tpl/polls/view_poll.html"
10    })
11    .state('editPoll', {
12      url: "/campaigns/:campaignId/polls/:pollId/edit",
13      templateUrl: "/assets/tpl/polls/editpoll.html",
14    })
15    .state('campaigns', {
16      url: "/campaigns",
17      templateUrl: "/assets/tpl/campaigns/campaigns.html"
18    })
19    .state('oneCampaign', {
20      url: "/campaigns/:campaignId",
21      templateUrl: "/assets/tpl/campaigns/edit_campaign.html"
22    })
23    .state('kpis', {
24      url: "/kpis",
25      templateUrl: "/assets/tpl/kpis/kpis.html"
26    })
27    .state('urlHandles', {
28      url: "/urlhandles",
29      templateUrl: "/assets/tpl/url_handles/url_handles.html"
30    });
31  });

```

Codice 5.3: Routing di pollcenter

Come si può notare dal codice, sono stati creati degli stati per le sezioni principali dell'interfaccia dell'applicazione, ovvero campagne, poll, kpi e URL. Ad esempio, quando nella barra degli indirizzi l'URL sarà `pollcenter.fake/campaigns`, **\$stateProvider** effettuerà il routing alla view generata dal template corrispondente e l'applicazione si troverà nello stato "campaigns". È importante considerare che non solo verrà compilata la view, ma anche il controller ad essa associata verrà istanziato:

```

1  <!-- campaigns.html -->
2
3  <div class="pc-cards-holder" layout="row" flex="100" layout-wrap layout-
4    padding ng-controller="CampaignsListController as campaignsCtrl">
5    <div stats-card class="pc-stats-card" title="{{campaign.name}}" subtitle
6      ="{{campaign.ts | date}}" trend="{{campaign.trend}}" data="campaign.stats
7      " view-state="oneCampaign({ campaignId : '{{campaign.id}}' })">
8    </div>
9    <div ng-repeat="campaign in campaignsCtrl.campaigns"></div>
10    <div ng-if="campaignsCtrl.campaigns.length == 0">
11      Non ci sono campagne, creane una usando il pulsante in basso!
12    </div>
13    <md-button class="md-fab md-fab-bottom-right pc-fab-campaign" aria-label="Add
14      " ng-click="campaignsCtrl.showDialog()">
15      <div class="pc-fab-icon-holder-campaign">
16        <i class="material-icons pc-fab-icon-campaign">add</i>
17      </div>
18    </md-button>
19  </div>

```

Codice 5.4: Template della view campaigns.html che verrà compilata a seguito del routing

Sarà possibile navigare tra gli stati anche programmaticamente, utilizzando il nome di uno stato e il service **\$state**, mediante il comando `$state.go(nomeDelloStato)`.

5.1.3 Campagne

La realizzazione del modulo delle campagne è stata relativamente semplice. Per prima cosa, si è reso necessario implementare il service e la sua interfaccia, in modo che controller e view potessero comunicare con il back-end dell'applicazione senza problemi. Tutti i service dell'applicazione utilizzano il service **\$http** fornito da AngularJS. Esso consente di effettuare chiamate HTTP ad un URL mediante i metodi *get()*, *post()*, *put()* e *delete()*. La struttura di una richiesta HTTP in Angular è la seguente

```
1 $http.get('someURL')
2 .then(function successCallback(response) {
3     // questa funzione verrà chiamata in modo asincrono
4     // quando sarà disponibile una risposta alla richiesta
5 }, function errorCallback(response) {
6     // questa funzione verrà chiamata in modo asincrono nel caso in cui
7     // avvenga un errore
8     // se il server ritorna un codice di errore
9 });
```

Codice 5.5: Esempio di richiesta HTTP

L'istruzione `$http.get('someURL')` restituisce una *promise*, ovvero un oggetto che consente l'esecuzione di codice in modo asincrono: quando la richiesta viene effettuata, l'applicazione non attende la risposta ma prosegue con l'esecuzione. Una volta che il server risponde, viene eseguito il blocco *then*. In esso sono tipicamente presenti due funzioni di callback: la prima viene invocata in caso di successo (risposta con stato HTTP compreso tra 200 and 299), la seconda in caso di fallimento (timeout del server o errore HTTP come 404).

Ecco come si presenta il service *campaignsService*:

```
1 angular.module('pollcenter.campaigns', [])
2 .service('campaignsService', [ '$http', function( $http){
3     var apiUrl = "/api/campaigns";
4
5     this.getCampaigns = function () {
6         return $http.get(apiUrl)
7         .then(function (data){
8             //in caso di successo restituisce l'oggetto JSON
9             return data.data;
10        }, function(data){
11            //in caso di errore restituisce lo status
12            return data.status;
13        });
14    };
15
16    this.getCampaign = function (campaignId) {
17        return $http.get(apiUrl + '/' + campaignId)
18        .then(function (data){
19            //in caso di successo restituisce l'oggetto JSON
20            return data.data;
21        }, function(data){
22            //in caso di errore restituisce lo status
23            return data.status;
24        });
25    };
26
27    this.saveCampaign = function (campaignData) {
28        if (campaignData.id !== undefined){
29            return $http.put(apiUrl + '/' + campaignData.id, campaignData)
```

```
30     .then(function (data){
31         //in caso di successo restituisce l'oggetto JSON
32         return data.data;
33     }, function(data){
34         //in caso di errore restituisce lo status
35         return data.status;
36     });
37 }
38 else {
39     return $http.post(apiUrl, campaignData)
40     .then(function (data){
41         //in caso di successo restituisce l'oggetto JSON
42         return data.data;
43     }, function(data){
44         //in caso di errore restituisce lo status
45         return data.status;
46     });
47 }
48 };
49
50 this.getCampaignVotesOverTime = function(campaignId){
51     return $http.get(apiUrl + '/' + campaignId + '/stats/votesvertime')
52     .then(function (data){
53         //in caso di successo restituisce l'oggetto JSON
54         return data.data;
55     }, function(data){
56         //in caso di errore restituisce lo status
57         return data.status;
58     });
59 };
60 }));
```

Codice 5.6: Implementazione di campaignsService

In caso di errore nella risposta del server, la view mostra un modale che indica all'utente il malfunzionamento temporaneo dell'applicazione. Successivamente sono stati sviluppati i template delle view ed i controller.

Il risultato finale è il seguente:

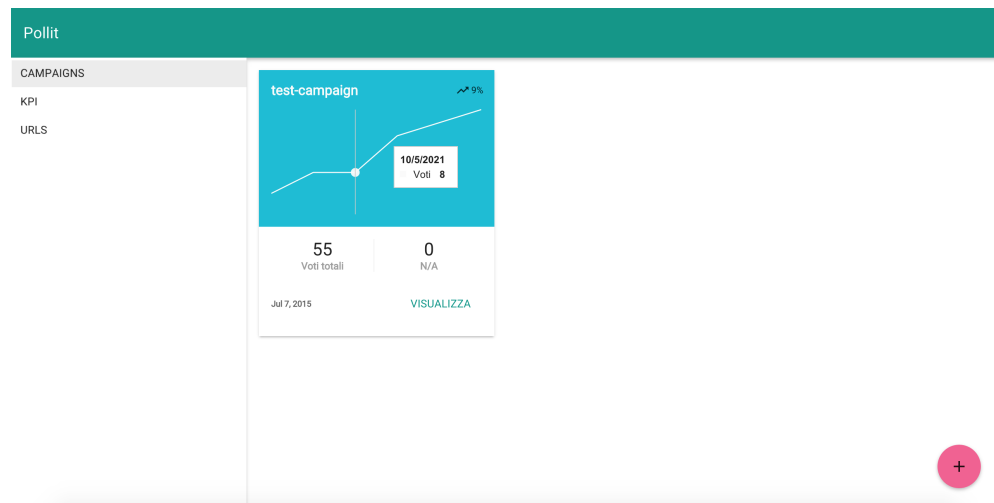


Figura 5.1: View della dashboard delle campagne di *pollcenter*

La dashboard utilizza le card del Material Design per dare visione delle campagne disponibili. Una card contiene indicazioni riguardo al numero di voti complessivo dei poll di una campagna ed un grafico che mostra il variare dei voti nel tempo. Utilizzando il bottone nelle card, il routing dell'applicazione porta alla view di dettaglio e di modifica di una campagna:

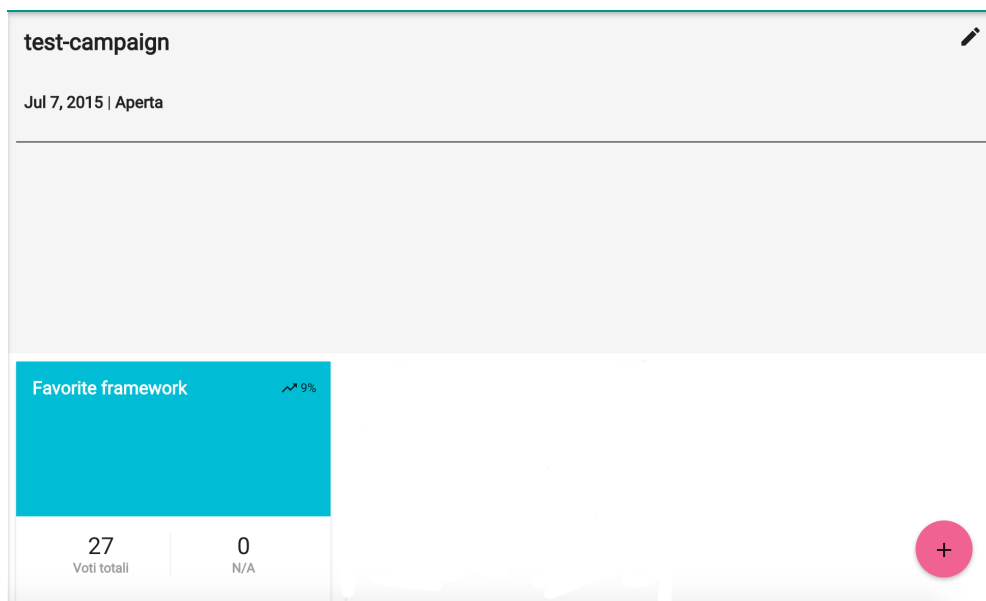


Figura 5.2: View di dettaglio e di modifica di una campagna

5.1.4 Poll

La realizzazione di questo modulo ricalca la procedura seguita nello sviluppo del modulo delle campagne. In questo modulo è stato però possibile sfruttare a pieno le potenzialità delle directive di AngularJS.

Al completamento della realizzazione, la view che consente di modificare un poll è risultata la seguente:

Informazioni sul poll

Nome
Favorite framework

Descrizione
Poll per scoprire qual'è il framework preferito dalla gente

Domanda

Domanda
Quale framework preferisci?

angular Ricerca KPI

angularJS

Figura 5.3: View di dettaglio e di modifica di una poll. Il FAB consente di aggiungere una domanda tra i tre tipi disponibili

Ogni poll può contenere domande di tipo rating, scelta testuale e scelta di immagini, che sono state realizzate mediante tre directive.

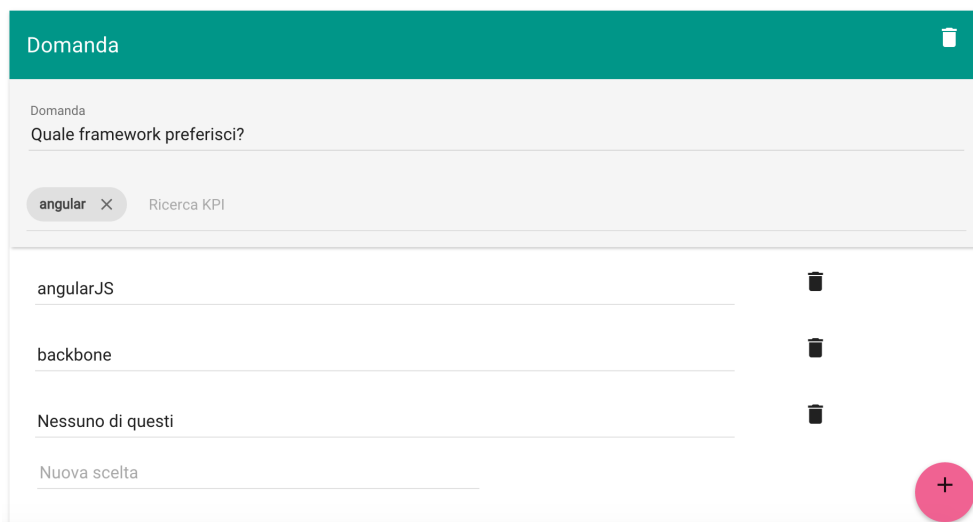
Domanda

Domanda
Valuta la semplicità di utilizzo di AngularJS

angular Ricerca KPI

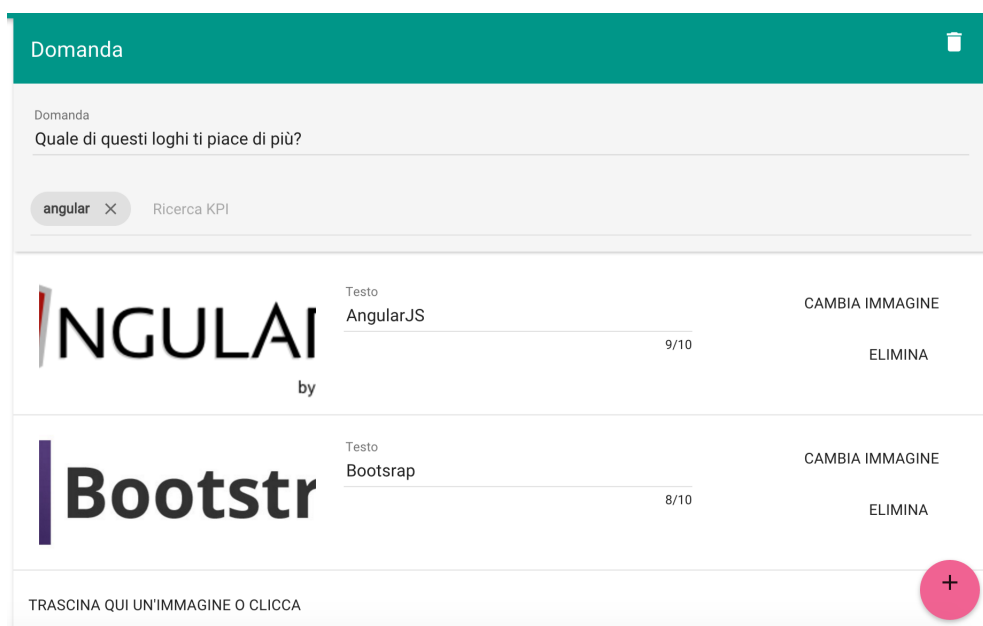
★★★★★ Questa domanda riceverà una valutazione da 1 a 5

Figura 5.4: La directive per le domande di tipo rating compilata nella view



The screenshot shows a web interface for a poll. At the top, there is a teal header bar with the word "Domanda" and a trash icon. Below this, the question "Domanda" is followed by "Quale framework preferisci?". A search bar contains "angular" with a close button and "Ricerca KPI". Below the search bar, there are three options: "angularJS", "backbone", and "Nessuno di questi", each with a trash icon to its right. At the bottom, there is a "Nuova scelta" input field and a pink circular button with a plus sign.

Figura 5.5: La directive per le domande di tipo scelta testuale compilata nella view



The screenshot shows a web interface for a poll. At the top, there is a teal header bar with the word "Domanda" and a trash icon. Below this, the question "Domanda" is followed by "Quale di questi loghi ti piace di più?". A search bar contains "angular" with a close button and "Ricerca KPI". Below the search bar, there are two image choices. The first choice shows the AngularJS logo with the text "Testo AngularJS" and a rating of "9/10". To the right of the logo are two buttons: "CAMBIA IMMAGINE" and "ELIMINA". The second choice shows the Bootstrap logo with the text "Testo Bootstrap" and a rating of "8/10". To the right of the logo are two buttons: "CAMBIA IMMAGINE" and "ELIMINA". At the bottom, there is a text input field with the placeholder "TRASCINA QUI UN'IMMAGINE O CLICCA" and a pink circular button with a plus sign.

Figura 5.6: La directive per le domande di tipo scelta di immagini compilata nella view

La directive che realizza le domande di tipo scelta di immagini deve poter consentire all'utente di caricare delle immagini, limitate alla dimensione a 2MB. L'azienda ha reso disponibile un server per l'upload mediante il servizio di Amazon AWS S3. Per realizzare l'upload è prima necessario ottenere una firma per l'immagine da caricare. Per questo scopo è stato creato un service ad hoc, **s3**, che espone un metodo che interroga le **API** di pollit e restituisce le informazioni necessarie per procedere con il caricamento dell'immagine sul server. Il team si è affidato al modulo di terze parti *ngFileUpload*.

```

1 s3.getImgChoiceUploadParams($scope.campaignId, $scope.pollId, $scope.questId)
2   .then(function (data){
3     params = data;
4     console.log(data.postArgs);
5     upload.upload({
6       url: params.action,
7       method: 'POST',
8       withCredentials: false,
9       fields: params.postArgs,
10      file: $scope.file
11    })

```

Codice 5.7: Upload di immagini

Il codice mostra che alla risoluzione della promise *getImgChoiceUploadParams*, viene restituito un oggetto contenente informazioni circa l'indirizzo del server S3 e i permessi di upload. Successivamente viene utilizzato il service **upload** del modulo *ngFileUpload* per procedere con l'operazione.

Per aumentare la separazione dei contesti nell'applicazione e per ottenere un codice della view molto snello, non sono state utilizzate le tre directive in modo diretto. Si è fatto ricorso ad un'ulteriore directive: *questEditorDirective*. Essa crea lo "scheletro" del layout di una domanda e integra la funzione di aggiunta dei kpi mediante tag (o chip in Material Design). Inoltre viene utilizzata inizializzando il parametro *quest* con l'oggetto JSON rappresentante la domanda. In questo modo essa può riconoscere il tipo della domanda e scegliere quale tipo di directive dovrà essere utilizzato per rappresentarla, aggiungendola al proprio template. Potrà anche modificare la domanda grazie al binding che detiene con l'oggetto a cui *quest* riferisce.

Quindi, quando verrà compilata la view di modifica e visualizzazione di un poll, verrà istanziato il controller collegato. Esso recupererà mediante i service dalle **API** l'array di domande create in precedenza per il poll, aggiungendolo al proprio scope. *questEditorDirective* potrà quindi accedere allo scope del controller, renderizzare le domande mediante componenti grafici del Material Design ed eventualmente modificare la domanda o le scelte disponibili. Il codice della view viene riportato in seguito. Si può notare quanto questo approccio lo abbia reso leggibile e di poche linee.

```

1 <div layout="column" ng-controller="EditPollController as editPollCtrl">
2   <!-- editPollCtrl contiene l array delle domande nel parametro editPollCtrl
3     .poll.questions-->
4     <div layout="row" flex="100" layout-align="center" layout-padding>
5       <div layout="column" flex="100" flex-gt-md="100" flex-md="100" style=
6         "margin-bottom: 100px;">
7
8         <!-- la directive viene ripetuta per ogni domanda presente nel
9           model, ovvero poll.questions, mediante ng-repeat-->
10        <div quest-editor quest="q" delete-quest="editPollCtrl.

```



```
8      deleteQuest(quest)" ng-repeat="q in editPollCtrl.poll.quests track by q.  
9      id" kpis="editPollCtrl.kpis" ></div>  
10    </div>  
11  </div>
```

Codice 5.8: Implementazione della view di visualizzazione e modifica di un poll

5.1.5 Kpi

La realizzazione del modulo che gestisce i kpi riproduce semplicemente quanto fatto per il modulo delle campagne. Inoltre, data la natura prototipale di *pollcenter*, non sono state ancora stabilite delle metriche e delle statistiche da mostrare.

5.1.6 URL

L'ultimo modulo implementato realizza le funzionalità di creazione e gestione degli URL da associare alle campagne. Anche in questo caso dopo aver realizzato i service necessari, si è passati all'implementazione dei template e dei controller. In particolare, la pagina degli URL mostra una lista con gli URL creati.

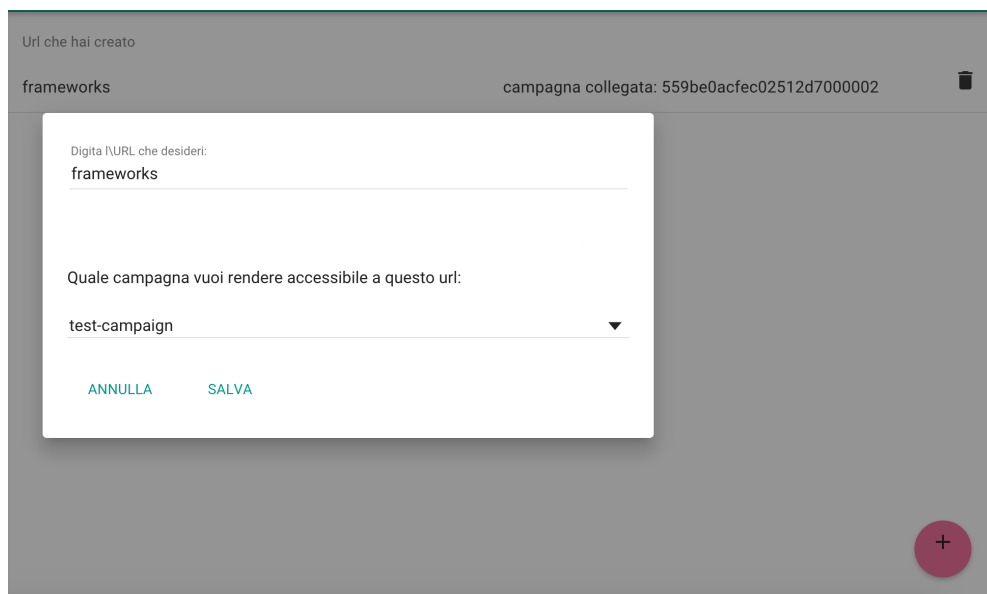


Figura 5.7: La creazione di un URL dall'applicazione

Attraverso il FAB button si ha accesso ad un dialog modale, nel quale viene richiesto di inserire un nome per completare l'URL da distribuire per utilizzare *pollboy* e la campagna a cui deve essere associato. Se si tenta di associare una campagna a più di un URL, nel modale apparirà un messaggio di errore. Nell'esempio riportato dall'immagine, supponendo che *pollboy* sia disponibile all'URL *pollboy.com*, i questionari per la campagna *test-campaign* saranno accessibili all'indirizzo *pollboy.com/frameworks*.

5.2 pollboy

La codifica e la realizzazione di *pollboy* si sono rivelate molto semplici in quanto le funzionalità da implementare erano poche e ben delineate.

È stata data particolare importanza all'interazione dell'utente con l'interfaccia, poiché si vuole consentire un'esperienza di utilizzo piacevole. Tipicamente, rispondere alle domande di un questionario risulta tedioso e a volte snervante. Dopo aver provato diverse *gesture* da impiegare nella selezione della risposta alle domande, il *tap* è risultato il più comodo ed efficace: è la *gesture* più comune nelle applicazioni per dispositivi mobili.

Al termine della realizzazione, l'interfaccia di *pollcenter* risulta completamente in Material Design e ricorda l'esperienza tipica che un utente passa utilizzando applicazioni per smartphone.



Figura 5.8: Una domanda a scelta testuale nell'interfaccia di pollboy

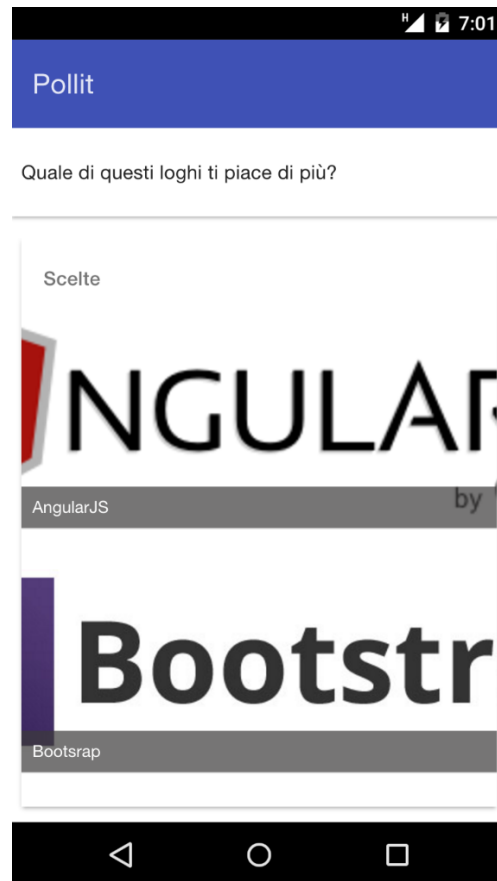


Figura 5.9: Una domanda a scelta di immagini nell'interfaccia di pollboy

Capitolo 6

Conclusioni

Il capitolo finale espone le conclusioni tratte riguardo alle attività svolte durante lo stage.

6.1 Valutazione dei risultati ottenuti

Il prototipo sviluppato rispecchia le aspettative dell'azienda. Gli sforzi per ottenere un rilascio di nuove funzionalità che lo rendano effettivamente un prodotto completo non saranno eccessivi, anche grazie all'ottima scalabilità ottenuta.

pollcenter e *pollboy* rispettano tutti i requisiti funzionali, obbligatori e desiderabili, delineati in [Analisi dei Requisiti](#). Per quanto riguarda i requisiti di vincolo, non sono rispettati i requisiti facoltativi **RVF7** “L'applicazione deve poter funzionare correttamente nel browser Internet Explorer versione 9 o superiore” e **RVF8** “L'applicazione deve poter funzionare correttamente nel browser Internet Explorer per dispositivi mobili Windows Phone versione 8 o superiore” sia per *pollboy* che per *pollcenter*. Questo è dovuto all'utilizzo del layout [flexbox](#) che presenta alcuni problemi di compatibilità con quei browser.

6.2 Aspetti critici nell'uso framework AngularJS

L'esperienza dello sviluppo con AngularJS è stata molto positiva ed ha portato diversi vantaggi. È un framework che fornisce molti strumenti avanzati per creare applicazioni in JavaScript e la sua struttura rende possibile pensare alla realizzazione di progetti anche piuttosto complessi. Basti pensare alla grande scalabilità che consente la suddivisione dell'applicazione in moduli.

AngularJS consente di scrivere codebase snelle e ben strutturate e di separare logicamente le componenti come controller e service dalle view, attuando una separazione dei contesti profonda e facilitando i test di unità. Inoltre l'implementazione della Dependency Injection e la conseguente istanziatura on-demand dei componenti facilita il caricamento dell'applicazione nel browser.

Oltre ad evidenti vantaggi, durante lo svolgimento del progetto sono emersi alcuni difetti e problemi. Innanzitutto, le basi nell'utilizzo del framework sono semplici da acquisire ma per poter applicare le best-practices e produrre applicazioni di qualità è necessario uno studio profondo delle funzionalità offerte, non acquisibile solamente mediante la documentazione. Altro punto debole è proprio la documentazione offerta

che, pur essendo completa, non riesce ad essere davvero efficace.

Il concetto di scope ed il two-way data-binding sono funzionalità comode e molto efficaci, tuttavia durante lo sviluppo alcune porzioni di codice sono state riscritte da zero in quanto l'annidamento degli scope aveva reso impossibile il debug degli errori. Nonostante ciò, AngularJS si candida come uno dei migliori e più usati framework JavaScript e attualmente esistono moltissimi moduli e librerie di terze parti che ne aumentano le funzionalità e le potenzialità.

6.3 Aspetti critici nell'uso del Material Design

L'impiego delle linee guida e dei componenti del Material Design ha permesso di ottenere un layout piacevole e completamente responsive. L'utilizzo delle interfacce è semplice ed immediato. I risultati migliori sono stati ottenuti provando l'applicazione da dispositivo mobile: la fase di apprendimento è risultata molto breve. Questo è reso possibile dall'esperienza utente del tutto simile a quella percepibile su svariate applicazioni native per smartphone e a cui in genere l'utente è abituato.

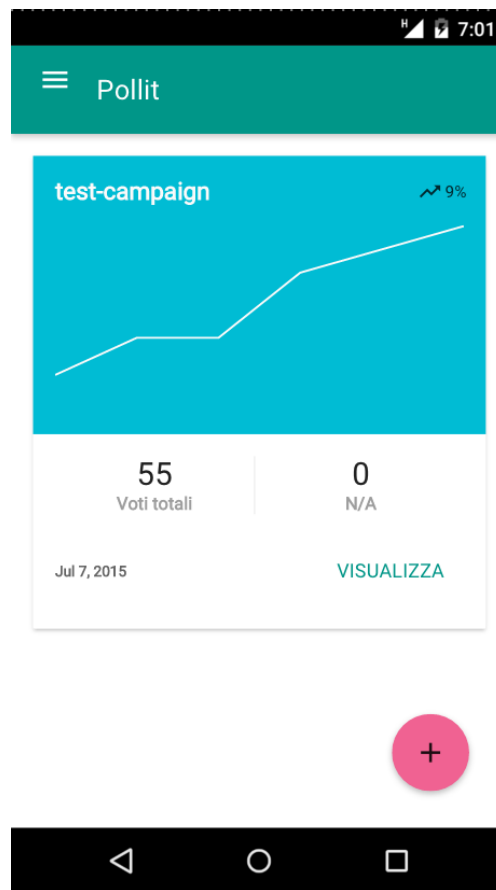


Figura 6.1: *pollcenter* su smartphone

Tuttavia vi sono alcuni riscontri negativi. Ad esempio, l'ampio utilizzo di ombre ed effetti di movimento appesantisce il caricamento dell'applicazione. In applicativi

complessi ciò potrebbe compromettere anche la fluidità dell'interfaccia. L'esperienza su dispositivi desktop è altrettanto semplice ed efficace ma, a causa dello stile minimale e flat del layout, in schermi ampi l'interfaccia può apparire sterile.

6.4 Conoscenze acquisite

Durante il periodo di stage sono state acquisite competenze riguardo lo sviluppo di applicazioni web che devono tenere conto in particolar modo dell'ambiente mobile, non solo dell'esecuzione in browser desktop. È stato possibile osservare l'importanza dell'esperienza utente nella progettazione di interfacce grafiche.

Prima dello stage il framework AngularJS era stato impiegato in maniera molto semplice in un progetto didattico. Le basi però non erano sufficienti alla produzione di applicazioni complesse e allo sviluppo di codice di qualità. Sono state inoltre rafforzate le competenze riguardo il linguaggio JavaScript, che era già noto prima dello stage.

Infine, utilizzando le [API](#) fornite per **Pollit** è stato possibile osservare come viene implementata un'architettura [REST](#) a livello aziendale e come un'applicazione che ne faccia uso debba essere strutturata.

Dal punto di vista dell'integrazione all'interno dell'azienda, ho potuto acquisire nozioni sull'organizzazione e sulle fasi del ciclo di vita di un software, oltre alla possibilità di capire come relazionarsi all'interno di un team in ambito lavorativo.

Glossario

APIs Indica un'insieme di procedure rese disponibili al programmatore allo scopo di ottenere un'astrazione della complessità della piattaforma sottostante, che può essere sia hardware che software. [2](#), [12](#), [17](#), [18](#), [25](#), [31](#), [32](#), [34](#), [36–38](#), [43–48](#), [51](#), [52](#), [60](#), [67](#), [69](#)

B2B Acronimo di *Business-to-Business*, in italiano *commercio interaziendale*. Indica le relazioni che un'impresa detiene con i propri fornitori per attività di approvvigionamento e di pianificazione e le relazioni che l'impresa detiene con altre imprese, collocate in punti diversi della filiera produttiva. [1](#), [69](#)

CRM Acronimo di *Customer Relationship Management*. In marketing indica la gestione dei rapporti con i clienti mediante strategie commerciali ed è legato al concetto di fidelizzazione. [1](#), [69](#)

CRUD Acronimo di *create, read, update and delete*. Fa riferimento alle principali operazioni implementate in una base di dati. Un'applicazione con operazioni CRUD è quindi in grado di creare, leggere, modificare ed eliminare dati persistenti. [31](#), [69](#)

DAM Acronimo di *Digital Assets Management*. Si riferisce alla gestione di processi che riguardano la catalogazione, la distribuzione e lo storage di asset digitali, ovvero file binari come immagini e testo. Spesso questi processi vengono gestiti attraverso l'uso di software. [1](#), [69](#)

DOM Il DOM o *Document Object Model* è lo standard del W3C per la rappresentazione ad oggetti di documenti strutturati, come le pagine HTML. [6](#), [8–12](#), [69](#)

Flexbox Acronimo di *Flexible box layout*. Il modello di layout flessibile consente di sistemare in maniera flessibile i box presenti all'interno di un elemento contenitore, offrendo vari metodi per gestione l'ordine di visualizzazione dei box, lo spazio disponibile e l'allineamento. Sarà necessario definire un elemento contenitore che avrà la proprietà CSS `display: box`. [15](#), [16](#), [65](#), [69](#)

Gesture Combinazione di movimenti dell'utente effettuati con le dita su un dispositivo touch-screen, che vengono riconosciuti da un'applicazione. [25](#), [47](#), [49](#), [62](#), [69](#)

Minificazione Processo che mira a ridurre la dimensione di un codice sorgente, rimuovendo da esso elementi non utili al compilatore (ad esempio tabulazioni e commenti). In ambito web è una prassi ricorrente e piuttosto utile, poichè

ridurre la dimensione dei file, comporta un minor tempo di caricamento per le pagine. Tipicamente la minificazione del codice viene effettuata con strumenti automatici. [7](#), [69](#)

MVC Pattern architetturale che prevede la separazione tra la logica di gestione dei dati e come questi dati vengono presentati. Il pattern prevede la divisione dell'architettura in tre parti: **Model**: si occupa della gestione dei dati; **View**: si occupa di visualizzare i dati presenti nel model; **Controller**: si occupa di aggiornare il model in base alle operazioni che l'utente compie sulla view. [2](#), [5](#), [29](#), [70](#)

npm Acronimo di Node Package Manager, è un sistema di gestione delle dipendenze per le applicazioni JavaScript che permette di installare librerie di terze parti mediante un'interfaccia a riga di comando. [16](#), [70](#)

REST Riferisce ad un insieme di principi di architetture di rete, i quali delineano come le risorse sono definite e indirizzate:

- *identificazione univoca delle risorse*: ad esempio, nel web devono essere identificate univocamente con un URI;
- *utilizzo esplicito dei metodi HTTP*;
- *risorse autodescrittive*: è possibile utilizzare virtualmente qualsiasi formato per rappresentare le risorse, ma è opportuno
- *utilizzare formati il più possibile standard in modo da semplificare l'interazione con i client*;
- *collegamenti tra risorse*: una risorsa deve fornire tutte le informazioni riguardo alle risorse ad essa correlate nella sua rappresentazione o mediante collegamenti ipertestuali;
- *comunicazione stateless*: nessuna richiesta deve avere relazioni con le richieste precedenti e successive ad essa.

Il termine è spesso usato per descrivere ogni interfaccia che trasmette dati mediante HTTP. [12](#), [32](#), [67](#), [70](#)

REST-Like Architettura software in stile REST, che discosta da questa non vincolandosi all'aderenza di tutti i principi definiti per REST. [2](#), [32](#), [70](#)

Singleton Il singleton è un design pattern ha lo scopo di garantire che venga creata una sola istanza di una determinata classe, e di fornire un punto di accesso globale a tale istanza.. [70](#)

Tap Gesture che consiste in un singolo tocco dello schermo da parte dell'utente, è l'equivalente di un click del mouse. [47](#), [49](#), [62](#), [70](#)

UX Acronimo di *User eXperience*. Indica le percezioni e le reazioni di un utente che derivano dall'uso di un prodotto. Spesso le interfacce utente vengono progettate cercando di suscitare emozioni o reazioni particolari. [2](#), [70](#)

Bibliografia

Angular Material. URL: <https://material.angularjs.org/latest/>.

Bower. URL: <http://bower.io/>.

Brad Green, Shyam Seshadri. *AngularJS: Up and Running*. O'Reilly Media, Inc.

Documentazione AngularJS. URL: <https://docs.angularjs.org>.

Introduzione a CSS3. URL: http://www.w3schools.com/css/css3_intro.asp.

Introduzione ad HTML5. URL: http://www.w3schools.com/html/html5_intro.asp.

Material Design. URL: <https://www.google.com/design/spec/material-design/introduction.html>.