Capitolo 1

Ruby On Rails

1.1 Ruby On Rails ~ RoR

In questo capitolo è trattato lo sviluppo di RBlog attraverso Ruby on Rails, nel seguito RoR, sfruttando il metodo ATDD, Acceptance Test-Driven Development.

Nella prima sezione è introdotto Ruby, linguaggio in cui è implementato RoR, di cui sono indicate le principali caratteristiche, introducendo elementi utili per favorire la comprensione del progetto.

Nella sezione 1.2 è introdotto RoR e la sua architettura, descrivendone le particolarità. Le sezioni dalla 1.3 fino al termine del capitolo trattano lo sviluppo dell'applicazione web RBlog attraverso esempi e frammenti di codice, descrivendo il processo di sviluppo di un'insieme di funzionalità, raggruppate in funzione degli strumenti utilizzati, e dell'implementazione dei relativi test di accettazione.

1.1.1 Ruby

Ruby è un linguaggio open-source e general-purpose, ideato da Yukihiro Matsumoto e rilasciato per la prima volta nel 1995 ed attualmente alla versione 2.1[1]. E' un linguaggio di programmazione orientato agli oggetti puro ed è tipato dinamicamente.

Listato 1.1: A differenza di Java ed altri linguaggi ad oggetti, ogni tipo è definito da una classe. Nell'esempio il metodo "next" è invocato su un'espressione di tipo "FixNum".

```
1 2.1.1 :002 > (1+1).next
2 => 3
```

Ruby è fortemente orientato alla produttività, permette di scrivere codice essenziale rispetto a linguaggi tipati staticamente come Java o C# e offre una sintassi ricca e versatile.

Listato 1.2: Dichiarazione della classe Rectangle.

```
1  class Rectangle
2    def initialize w,h
3      @w = w
4      @h = h
5    end
6    def area
7      @w*@h
8    end
9  end
10 r = Rectangle.new 2, 4.5
11 puts r.area
12
```

La dichiarazione di una classe incomincia con la keyword "class" seguito dall'identificatore ed è conclusa con il terminatore "end". Per creare un nuovo oggetto è utilizzato il metodo "new" che inizializza lo stato degli attributi invocando "initialize", ad esempio nello script è creato un nuovo rettangolo "r" specificando larghezza ed altezza.

Gli attributi d'istanza non sono dichiarati a priori ma dinamicamente, come avviene in Python, e sono acceduti tramite il token "@". Gli attributi di classe si accedono tramite "@@".

Ruby offre una sintassi minima per favorire la leggibilità del codice, ad esempio non è obbligatorio l'uso di parentesi e i metodi restituiscono il valore dell'ultima espressione eseguita, come il metodo "area".

Listato 1.3: Estensione della classe Rectangle.

```
1  #...
2  class Rectangle
3    attr_accessor :color
4  end
5
6  r.color = 'Red'
7  puts r.color
8  puts r.area
9
10 =>Red
11 =>9.0
```

Un'altra caratteristica interessante è la modificabilità delle classi. Il frammento 1.3 mostra come la dichiarazione di una classe con identificatore già esistente permetta di far variare il comportamento della precedente versione. Gli oggetti già esistenti sono aggiornati e possono reagire correttamente alle nuove richieste, ad esempio il rettangolo "r", già dichiarato ed istanziato, possiede adesso una coppia di metodi getter e setter per l'accesso all'attributo pubblico "color" mantenendo anche le precedenti funzionalità, come il metodo "area".
L'elemento ":color", utilizzato per creare i metodi d'accesso alla colorazione del rettangolo, è un simbolo[2]. A differenza delle stringhe[3], i simboli in Ruby sono oggetti immutabili ed unici, possono essere generati dinamicamente e spesso

sono utilizzati come riferimenti, ad esempio ad attributi o metodi, oppure come chiavi nelle strutture dato, in quanto la comparazione è completata in O(1).[1]

Listato 1.4: Dichiarazioni di due dizionari equivalenti, utilizzando una diversa sintassi.

```
1 options = { :font_size => 10, :font_family => "Arial" }
2 options2 = { font_size: 10, font_family: "Arial" }
```

Nell'esempio sono dichiarati due dizionari contenenti le stesse coppie simbolo-valore, ma definiti attraverso due sintassi diverse. Frequentemente i metodi in Ruby utilizzano i dizionari e i simboli per specificare in un solo parametro un insieme di opzioni. Sfruttando una delle sintassi dell'esempio si istanza un oggetto di tipo "Hash"[5], ma è anche possibile utilizzare esplicitamente i costruttori disponibili ed ottenere lo stesso risultato.

In Ruby esiste un garbage collector e non è previsto alcun meccanismo esplicito per la gestione della memoria. Gli oggetti, incluse le variabili locali, sono mantenuti sullo heap e non sullo stack.[6]

Ereditarietà

Listato 1.5: Definizione di una classe Square figlia di Rectangle.

```
1 class Square < Rectangle
2   def initialize s
3     super s, s
4   end
5 end
6 square = Square.new 2
7 puts square.area
8
```

In Ruby è possibile definire ereditarietà singola e, come in Java, è presente una gerarchia di tipi: ogni classe estende "Object"[7] che a sua volta estende "BasicObject"[8], la radice della gerarchia.

Listato 1.6: Navigazione della gerarchia delle classi.

```
1 puts Rectangle.superclass
2 => Object
3 puts Object.superclass
4 => BasicObject
5 puts SimpleObject.superclass
6 => nil
```

I moduli in Ruby permettono di raggruppare metodi, classi e costanti e definire dei namespace. A differenza delle classi sono non hanno uno stato e non possono essere istanziati.

[1] Dalla prossima versione di Ruby[4], la 2.2, sarà introdotto il symbol garbage collector. Al momento i simboli generati non rilasciano mai la memoria allocata introducendo potenziali memory leak.

Listato 1.7: Definizione di un nuovo modulo.

```
module Greeter
  def greet salute = 'Hi'
    puts salute
  end
end
```

Inoltre i moduli sopperiscono alla mancanza dell'ereditarietà multipla grazie ad una tecnica chiamata "mixin". Includendo i moduli all'interno di classi o altri moduli esistenti si estendono le funzionalità disponibili, permettendo la creazione di codice modulare e facilmente riutilizzabile.

Listato 1.8: Mixin dei moduli "Greeter" e "EnthusisticGreeter", per convenzione le costanti sono dichiarate utilizzando lettere maiuscole.

```
module EnthusisticGreeter
  include Greeter
  HYPE = 2
  def greet_with_enthusiasm
    HYPE.times do
      greet 'Hello!!!'
    end
  end
end

class Person
  include Greeter
  include EnthusisticGreeter
end

bob = Person.new
bob.greet
=> Hi
bob.greet_with_enthusiasm
=> Hello!!!
=> Hello!!!
```

I blocchi

Una delle caratteristiche più interessanti di Ruby è la possibilità invocare i metodi fornendo una closure tramite un blocco.

Listato 1.9: Le sintassi disponibili per i blocchi.

```
array = [1,2,3,4,5]
array.each {|i| puts i}
array.each do |i|
  puts i
end
```

I blocchi, posizionati dopo l'ultimo parametro del metodo, possono essere definiti da una coppia di parentesi graffe oppure con la sintassi "do ... end", per convenzione si utilizza quest'ultima versione per definire funzioni con più di un'istruzione. Il metodo "each" dell'esempio itera sull'array dichiarato nella

prima istruzione ed esegue il blocco per ogni valore presente.

I metodi nelle librerie di sistema di Ruby, soprattutto nelle funzionalità riguardanti le strutture dato, spesso offrono all'utente la possibilità di includere un blocco per personalizzarne il comportamento.

Listato 1.10: Implementazione di un metodo che esegue il blocco, se presente.

```
1 def try
2   if block_given?
3     yield(1, 2, 3)
4   else
5     "no block"
6   end
7 end
```

Per ciascuna invocazione è possibile definire al più un blocco; tramite il metodo "block_given?" è possibile verificare la presenza di una closure ed eventualmente eseguirla tramite l'istruzione "yield", nell'esempio seguita da tre parametri interi.

I blocchi non sono elementi di prim'ordine del linguaggio, non è quindi possibile attribuirne il valore ad una variabile, ma è permesso effettuare una conversione ad un oggetto lambda per ottenere un riferimento.

Strumenti per l'apprendimento di Ruby

La comunità di Ruby contribuisce in maniera attiva all'individuazione di bug e al miglioramento continuo del linguaggio. Di seguito sono forniti alcuni riferimenti utili per l'apprendimento.

I Ruby Koans[9], "koan" è la pronuncia giapponese dei caratteri cinesi, sono una raccolta di esercizi su Ruby e permettono lo studio del linguaggio comprendendo esempi guidati per apprendere la grammatica, le convenzioni e far pratica con le strutture dato. L'utente avanza nell'apprendimento in puro stile TDD, in maniera semplice e graduale. Terminato il corso si acquisisce un buon livello di confidenza con Ruby.

Sempre nello stile tracciato dai koan, tenuti in massima considerazione all'interno della comunità di Ruby, ho approfondito la conoscenza del linguaggio e di RoR tramite il portale Ruby Monk[10] che fornisce corsi di diverse difficoltà, comprendendo numerosi tutorial interattivi ed esercizi riassuntivi; ogni lezione è completabile attraverso il proprio browser internet e fornisce numerosi consigli e indicazioni allo studente.

1.1.2 RubyMine

Per lo sviluppo di RBlog e la definizione dei test di accettazione è stato utilizzato RubyMine[11], alla versione 6.3. L'IDE prodotto da JetBrains, sviluppatori

anche di IntelliJ IDEA, Android Studio, ReSharper per citare i prodotti più conosciuti, supporta le feature più recenti di Rails e Ruby.

L'esperienza con RubyMine è stata ottima: durante lo sviluppo sono stati rilevati raramente problemi, le funzionalità a riga di comando offerte da Rails sono integrate perfettamente e sono supportati molti linguaggi, come HTML, JavaScript, JQuery, CoffeeScript, SCSS.

Nell'IDE sono presenti diversi plugin per l'integrazione con strumenti di terzi, come ad esempio Cucumber, Git[2] e SSH. RubyMine è un prodotto curato nei dettagli, professionale ed allo stesso tempo adatto anche agli utenti alle prime armi; permettendo l'implementazione di un'applicazione in Rails praticamente senza abbandonare l'ambiente di sviluppo.

1.2 L'interpretazione di RoR del pattern MVC

Di seguito sono descritte le componenti principali del pattern architetturale MVC in funzione dell'interpretazione data da RoR e gli strumenti utilizzati per l'implementazione. E' anche presente una sezione riguardante il testing, in diverse forme, e come sia integrato all'interno del framework e dell'ambiente di sviluppo RubyMine.

1.2.1 Il modello

L'interpretazione di RoR del modello dell'architettura MCV prende spunto dal patter Active Record[12] introdotto da Martin Fowler. I record introducono un livello di astrazione fra i dati mantenuti nel modello ed i controlli che gestiscono e manipolano il dominio; le tuple nei database sono rappresentate da oggetti aventi sia le informazioni che li caratterizzano, ad esempio i diversi attributi e le relazioni con altri tipi di dato presenti nel dominio, sia i metodi d'istanza che ne descrivono il comportamento.

Non si accede direttamente al database, e in generale a qualunque sistema garantisca la persistenza della nostra applicazione, ma tramite l'interfaccia dell'ORM, Object-Relational Mapping. Questa tecnica minimizza, se non addirittura elimina, la necessita di eseguire codice nativo, come query SQL, per manipolare il dominio. L'uso del pattern Active Record permette a RoR di delineare in maniera netta la separazione fra modello e controlli.

Il modulo ActiveRecord di RoR fornisce molte funzionalità che estendono la rappresentazione ad oggetti del dominio con relazioni di ereditarietà fra i tipi esistenti, permettono la validazione attraverso l'invocazione di metodi e la definizione interrogazioni attraverso una libreria di sistema.

[2]Sono inclusi diversi plugin per il supporto sistemi di versionamento: attualmente sono disponibili CVS, Git, Subversion, Mercurial e Perforce.

Listato 1.11: Esecuzione di una query in Ruby che restituisce tutti gli articoli esistenti, dal più recente al più vecchio.

```
@posts = Post
      .where('title like ?', "#{params[:search]}%")
      .order('updated_at DESC')
```

1.2.2 I controlli

Un controllo in RoR[13] è rappresentato da una classe che estende "ActionController::Base"[14] e fornisce dei metodi pubblici a cui corrispondono le richieste soddisfacibili dall'applicazione.

Listato 1.12: Frammento del controllo dei Post.

```
class PostsController < ApplicationController
  # GET /posts
  # GET /posts.json
  def index
    if params[:search].present?
      @posts = Post
        .where('title like ?', "#{params[:search]}%")
        .order('created_at DESC')
    else
      @posts = Post.all.order('created_at DESC')
    end
  end

```

Nel listato 1.12 è visibile il controller dei Post ed il metodo "index", che si occupa di interagire con il modello per caricare gli articoli da visualizzare nel browser per la pagina principale del blog; la convenzione in RoR associa a ciascun metodo una vista con nome uguale.[3]

Per semplificare la definizione ed il mantenimento del codice dei controlli sono presenti alcuni accorgimenti. La configurazione delle richieste instradabili, è specificata in file separato "*routes.rb*"[15], all'interno del quale è possibile configurare le richieste HTTP che possono essere soddisfatte e definire una gerarchia delle entità manipolate dall'applicazione secondo i principi delle architetture REST, REpresentational State Transfer.

Listato 1.13: Frammento del file "routes.rb" relativo ai controlli di post e sessioni.

```
resources :posts do
  get :autocomplete_title, :on => :collection
end
resources :sessions, :only => [:new, :create, :destroy]
```

Nel frammento del file di configurazione dell'instradamento è specificata un'ulteriore azione per l'entità Post, oltre a quelle standard, e le tre azioni

[3] PostController estende ActionController::Base tramite la classe ApplicationController.

utilizzate per implementare i meccanismi di autenticazione.

Un altro accorgimento apprezzato durante lo sviluppo è la possibilità di definire dei filtri[16] per i controlli. Per ogni azione dei controlli sono previsti alcuni stati[4] a cui è possibile attribuire delle callback. Sfruttando questa semplice tecnica è possibile fattorizzare alcuni comportamenti comuni all'interno di un controllo.

Listato 1.14: L'uso dei filtri in RBlog.

```
1 before_action :require_login,
2   only: [:new, :create, :edit, :update, :destroy]
```

Nell'implementazione di RBlog è stato definito un filtro per verificare che non sia possibile compiere operazioni sensibili senza aver compiuto l'autenticazione. Grazie al metodo "before_action" è possibile eseguire codice definito dall'utente prima di eseguire una delle azioni specificate.

Listato 1.15: Verifica delle credenziali di accesso.

```
1 def require_login
2   unless current_user
3     redirect_to :log_in, :notice => 'Effettua il login prima.'
4   end
5 end
```

"current_user" è un metodo definito all'interno della classe "ApplicationController", quindi disponibile in tutti i controlli dell'applicazione, che verifica all'interno della sessione HTTP criptata la presenza di un attributo corrispondente all'identificatore univoco di uno degli autori di RBlog.

1.2.3 Le viste

Le viste in RoR sono definibili introducendo nei file HTML espressioni scritte in Ruby da processare all'interno del server web prima di fornire la risposta al client utilizzando ERB, Embedded RuBy.

ERB

Le viste in RoR coincidono con file con estensione ".html.erb"[5]. Un documento ERB estende le pagine HTML classiche, aggiungendo la possibilità di scrivere codice Ruby per definire dei comportamenti dinamici.

Listato 1.16: Hello ERB!

```
1 <html>
2   <body>
```

[4] Paragonabili alla annotazioni *@After*, *@Before*, *@AfterClass* e *@BeforeClass* in JUnit 4.

[5] La convenzione relativa alla struttura del progetto suggerisce di creare all'interno della cartella "views" tante cartelle quanti sono i controlli presenti nell'applicazione, più una cartella "layout" che conterrà le porzioni di viste condivise. I file dovrebbero essere organizzati secondo lo schema "views/nome_del_controllo/azione.html.erb".

```
3       <p>Hello, <%= user.first_name %>.</p>
4     </body>
5   </html>
```

All'interno di pagine HTML, codice JavaScript e JQuery è possibili introdurre i delimitatori di ERB. Nel breve esempio i delimitatori "<%= ... %>" contengono al loro interno un'espressione Ruby, il cui valore sarà valutato e concatenato al testo "Hello, " presente all'interno del tag HTML "p".

Listato 1.17: Altri delimitatori ERB.

```
1  <% if condition %>
2  <div>
3    <%= expression %>
4    <%# buggy_expression %>
5  </div>
6  <% end %>
```

Nella breve vista è utilizzata la sintassi "if-end", in generale è possibile utilizzare ogni costrutto e funzionalità, ed il tag HTML sarà processato se e solo se la condizione risulterà vera. La definizione di viste dinamiche[18] risulta immediata, è solo necessario apprendere le funzionalità dei pochi tag esistenti: i delimitatori "<% ... %>" valutano l'espressione senza visualizzarne il valore, la coppia "<%# %>" rappresenta un commento.

Le viste scritte tramite ERB sono facilmente leggibili grazie alle caratteristiche di Ruby[6] e alle numerose funzionalità presenti negli Helper e nelle librerie del linguaggio.

Listato 1.18: Frammento di vista relativo alla visualizzazione di un singolo post.

```
1  <div class='post'>
2    <p class='post_title'>
3      <%= link_to @post.title, @post %>
4    </p>
5    <p class='post_detail'>
6      <%= author_detail(@post) %>
7      </br>
8      <% post_details(@post).each do |detail| %>
9        <%= detail %>
10     <% end %>
11   </p>
12     <p class='post_content'>
13     <%= @post.body %>
14   </p>
15   ...
16 </div>
```

La vista parziale descrive la visualizzazione di un div contenente le informazioni di un singolo post. Il metodo "link_to" è definito nel modulo UrlHelper[20]

[6] Ogni espressione ha un valore. Il valore di ritorno delle funzioni e dei metodi è dato dall'ultima espressione eseguita. Le parentesi sono opzionali ed anche i parametri possono esserlo.

e fornisce le funzionalità per definire elementi utili alla navigazione web, nell'esempio è utilizzato per generare un collegamento al singolo post con il testo equivalente al titolo.

1.2.4 Il testing

Tramite RoR è possibile gestire l'intero stack di un'applicazione web, test inclusi[21]. Sfruttando il framework RSpec[22], standard de-facto in Ruby, è possibile testare il modello ed anche verificare i propri controller, definendo i parametri HTTP e i dati. RSpec fornisce molte funzionalità; un'analisi più approfondita è effettuata nelle sezioni relative all'implementazione dei test di accettazione.

Da Ruby 1.9 è anche incluso la libreria MiniTest[23], che fornisce le funzionalità per arricchire i propri test con delle callback da applicare a differenti stati dell'esecuzione della libreria di test, introduce oggetti "*mock*", consente di effettuare misurazioni delle prestazioni e molto altro.

Esistono numerose librerie sviluppate da terze parti che, dovendo sviluppare un'applicazione web più complessa di RBlog, aggiungerebbero numerose funzionalità semplificando il processo di testing, ad esempio DatabaseCleaner[24] è molto diffusa per la pulizia dei database utilizzati, così come di Factory Girl[25] per la popolazione del modello.

E' anche previsto che il modello sia presente in tre versioni, -test, sviluppo e produzione- al fine di concedere allo sviluppatore la libertà di eseguire test sulle nuove funzionalità senza dover effettuare continui backup dei dati ed assumere altre precauzioni.
RoR fornisce inoltre il tool a linea di comando Rake[26] per poter gestire al meglio l'esecuzione selettiva delle proprie librerie di test.

1.2.5 Peculiarità

L'obbiettivo di questa tesi non è l'uso approfondito di RoR e delle sue funzionalità, ma durante l'implementazione del blog sono state notate alcune peculiarità del framework che hanno contribuito affinché lo sviluppo si svolgesse in maniera lineare, concentrando l'attenzione sui test di accettazione piuttosto che a problematiche di contorno.

Come in altre piattaforme, in cui sono presenti strumenti per supportare la compilazione e la risoluzione delle dipendenze, Ruby introduce RubyGems[27], un package manager simile a Maven[28] e Gradle[29] per il mondo Java, per la gestione delle gemme. Ogni gemma rappresenta una libreria ed è definita attraverso nome, versione ed architettura di riferimento.

Ogni applicazione in RoR è caratterizzata da un Gemfile[30], un semplice script in Ruby che rappresenta l'insieme delle dipendenze del progetto. E' possibile indicare quali librerie includere in funzione del tipo di compilazione

adottata, rilascio, sviluppo o test, e delegando la verifica di aggiornamenti per librerie di terze parti al sistema.

Listato 1.19: Frammento del Gemfile di RBlog.

```
source 'https://rubygems.org'

group :development, :test do
  gem 'cucumber-rails', :require => false
  gem 'rspec-rails'
  gem 'capybara'
  gem 'poltergeist'
  gem 'database_cleaner'
end
```

Fra le gemme utilizzate, la più utile è stata Spring[31]. La libreria permette di mantenere l'applicazione in esecuzione in background, monitorando le modifiche effettuate al codice del progetto.

Spring utilizza un meccanismo di RoR per l'aggiornamento a run-time delle classi del progetto, mantenendo in esecuzione la versione più recente dell'applicazione ed evitando di dover riavviare l'esecuzione manualmente ad ogni cambiamento.

Nel contesto di sviluppo di applicazioni MVC è facile introdurre delle discrepanze fra gli schemi degli strumenti di persistenza e la rappresentazione delle entità all'interno dell'applicazione, soprattutto sfruttando strumenti di versionamento che offrono operazioni equivalenti alla "*revert*" in Git[32].

Listato 1.20: Migrazione relativa all'introduzione dell'entità Post nel modello.

```
class CreatePosts < ActiveRecord::Migration
  def change
    create_table :posts do |t|
      t.string :title
      t.text :body
      t.timestamps
    end
  end
end
```

In RoR è previsto il meccanismo delle migrazioni[33] per mantenere gli schema consistenti con il processo di sviluppo. Ogni variazione alla struttura del modello è tradotto in una migrazione, un breve script in Ruby, in cui sono definite le operazioni compiute a basso livello, come l'aggiunta di colonne, la rimozione di un vincolo etc.; le informazioni sulle migrazioni sono mantenute all'interno di una tabella, presente in tutti i database dell'applicazione, e tengono traccia dei cambiamenti apportati e della versione dello schema del dominio, permettendo di mantenere tutte le componenti dell'architettura MVC consistenti fra loro.

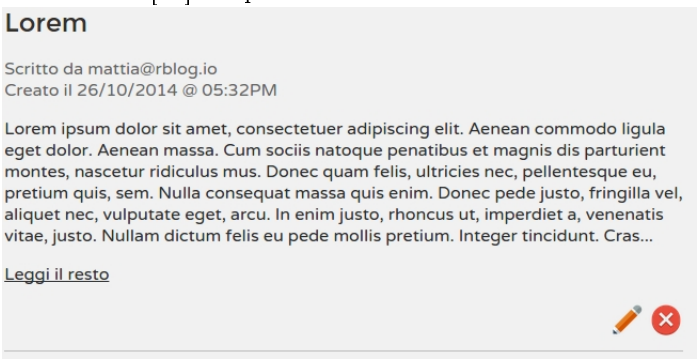
Don't Repeat Yourself ~ DRY

Il principio di mantenere il proprio codice senza ripetizioni e ben fattorizzato è un'ottima pratica di programmazione. Un progetto DRY permette agli svilup-

patori di modificare il codice più semplicemente; una funzionalità descritta in un numero ridotto di unità di compilazione e ben fattorizzata è facilmente individuabile, correggibile e modificabile senza incorrere in modifiche involontarie ad altre parti del sistema.

RoR fornisce differenti funzionalità, quali librerie, helper e viste parziali, per aiutare lo sviluppatore a definire applicazioni DRY.

Nell'implementazione delle viste è comune che alcuni elementi siano presenti più volte all'interno della stessa applicazione o anche all'interno della stessa pagina. In RoR è possibile definire delle viste parziali per fattorizzare al meglio queste porzioni di codice. Per rendere ancora più efficace questo meccanismo di fattorizzazione e definire dei comportamenti dinamici, è possibile passare al metodo "render"[34] dei parametri.



Listato 1.21: Utilizzo di una visita parziale con il metodo "render".

```
1 <%= render
2   :partial => 'posts/logged_user_post_actions',
3   :locals => {
4     :current_user => current_user,
5     :post => post
6   }
7 %>
```

Il codice ERB del listato 1.21 è utilizzato dalla vista per la lettura di un singolo post e dalla vista per la visualizzazione delle anteprime di tutti gli articoli presenti sul blog. Nell'esempio è passato al metodo "render" il nome della vista parziale da espandere e alcuni parametri.

Listato 1.22: Vista parziale contenente alcune operazioni su un singolo post.

```
1 <% if current_user %>
2   <div class='post_actions'>
3     <%= edit_post_image_link post %>
4     <%= remove_post_image_link post %>
5   </div>
6 <% end %>
```

La vista parziale verifica l'autenticazione del visitatore, se la variabile "current_user" ha valore nullo il codice non è eseguito, ed eventualmente visualizza

due immagini contenenti i collegamenti alle pagine di modifica e cancellazione del post. "edit_post_image_link" e "remove_post_image_link" sono due metodi ausiliari definiti all'interno del modulo "ApplicationHelper" di RBlog.

La piattaforma RoR fornisce anche numerosi Helper: moduli Ruby con funzionalità per il supporto dello sviluppo. Gli Helper in RoR sono sfruttati da tutte le componenti dell'architettura MVC e sono utili per gestire problematiche classiche dello sviluppo di applicazioni web: la gestione di file, formattazione di date, gestione di form e generazione del relativo codice HTML sono solo alcuni dei problemi risolvibili tramite gli helper presenti.

Listato 1.23: Il metodo 'truncate', appartenente al modulo TextHelper.

```
1 <%= truncate(post.body, :length => 500, :separator => ' ', :
      omission => '...') %>
```

Come già osservato le librerie di sistema sono ricche di parametri opzionali, spesso permettono l'uso di blocchi definiti dall'utente e sono ben documentate: ogni metodo presente è ampiamente descritto anche tramite esempi.

Listato 1.24: Un esempio dell'uso delle funzionalità di FormHelper.

```
1 <%= form_for(@post) do |f| %>
2   ...
3   <div id="title_field">
4   <%= f.label :title, 'Titolo' %>
5   ...
6   <%= f.text_field :title, :size => 50 %>
7   </div>
8   ...
9 <% end %>
```

E' altrettanto semplice definire i propri Helper: per ogni Controller definito dall'utente è incluso di default il relativo Helper, ad esempio in RBlog la classe PostsController include il modulo personalizzato PostsHelper in maniera automatica, senza necessità di configurarne l'uso e la visibilità all'interno del progetto.

Convention Over Configuration

RoR definisce un insieme di convenzioni per semplificare l'uso e la configurazione della piattaforma da parte dello sviluppatore. Per minimizzare il tempo richiesto per la messa a punto di un nuovo progetto, è utile seguire le convenzioni proposte, ordinando il codice in cartelle secondo le diverse funzionalità, aderendo alle convenzioni di denominazione dei file e degli attributi presenti nel modello.

Ovviamente non è possibile prescindere completamente dall'uso di alcuni file di configurazione, ma RoR facilita ulteriormente la definizione di questi file codificandoli con YAML[35], un formato di serializzazione facilmente leggibile e scrivibile, e Ruby stesso.

Il listato 1.25 descrive la configurazione della connessione ai database PostgreSQL[36] utilizzati per lo sviluppo di RBlog: la prima parte del documento riguarda la dichiarazione dei parametri standard, mentre i valori che seguono l'elemento "development" sono specifici del database per lo sviluppo. Il documento originale continua elencando i parametri per la connessione al database per i test.

Listato 1.25: Frammento in YAML relativo alla configurazione del modello.

```
1  default: &default
2  adapter: postgresql
3  encoding: unicode
4  host: localhost
5  pool: 5
6
7  development:
8  <<: *default
9  database: rblog_development
10 username: xblog
11 password: ...
```

Lo sviluppatore inoltre può anche sfruttare i numerosi tool a linea di comando per generare il codice relativo a viste, controlli e modello, ma anche file di configurazione, utili per ottenere una bozza contenente le impostazioni più plausibili per una nuova applicazione.

1.3 Hello RBlog!

L'obbiettivo della prima funzionalità "Hello RBlog!" è minimo, è necessario che l'applicazione sia in esecuzione ed il sito web raggiungibile. Inoltre il blog deve consentire la navigazione verso alcune pagine statiche.[7]

1.3.1 Cucumber

Il primo tassello scelto per l'implementazione dei test di accettazione su Rlog è Cucumber[37].

Cucumber è un framework per il supporto al BDD, Behaviour Driven Development, e la definizione di test di accettazione. Il framework è disponibile anche in Java, .Net, Python, PHP e molti altri linguaggi e piattaforme.

Le funzionalità

Listato 1.26: La prima feature di RBlog

```
1 @cap1
2 Funzionalità: Hello RBlog!
3 Per leggere i post e visitare il blog
4 Come Lettore
5 Vorrei che RBlog permettesse la navigazione
```

Il primo obbiettivo dei test di accettazione è di individuare le funzionalità da sviluppare, le caratteristiche e potenzialità che l'applicazione offre e i diversi scenari che le definiscono.

Le funzionalità in Cucumber sono elementi esclusivamente descrittivi, non sono utilizzate esplicitamente nell'implementazione dei test ma hanno lo scopo di far cogliere la giusta prospettiva al team di sviluppo e descrivere genericamente l'obbiettivo degli scenari. Sia il titolo che la descrizione possono essere in linguaggio naturale e non hanno alcun impatto nell'implementazione degli scenari, per lo sviluppo di RBlog sono state usate le user story[38] secondo il formato "*Per* <beneficio>, *come* <ruolo>, *vorrei* <obbiettivo, desiderio>".

Non ho capito la correzione "Perchè in italiano?"

Per facilitare lo sviluppo di test all'interno del proprio progetto è consigliato seguire la convenzione di inserire all'interno della cartella "*features*" i test di accettazione ed attribuire ai relativi file l'estenzione "*.feature*".

Nella funzionalità 1.26, oltre alla descrizione della funzionalità è presente un'etichetta che permette di organizzare e categorizzare quanto sviluppato con Cucumber; "*@cap1*" caratterizza sia la user-story sia gli scenari che la definiscono per ereditarietà. L'etichetta è stata introdotta per poter eseguire in maniera selettiva gli scenari della prima funzionalità.

[7] Una pagina web statica è una pagina web i cui contenuti sono formattati direttamente in HTML, e non subiscono modifiche in funzione dello stato attuale dell'applicazione. Al contrario le pagine dinamiche rappresentano lo stato di una o più entità presente all'interno del sito e possono variare nel tempo. In RBlog un esempio di pagina statica è la pagina che descrive l'abstract del progetto, mentre una pagina dinamica è la pagina che mostra un singolo post, ed ovviamente cambia nel contenuto in funzione dell'articolo scelto.

Listato 1.27: Comando per l'esecuzione degli scenari relativi alla feature "Hello RBlog!"

```
1 cucumber --tags @cap1
```

E' possibile introdurre un numero arbitrario di etichette per ciascuna funzionalità o scenario e, sfruttando l'opzione "*–tags*", definire il corretto insieme di test da eseguire, tramite gli operatori logici AND, OR e NOT.

Gli scenari

Listato 1.28: Navigazione verso la pagina iniziale

```
1 Scenario: Visita alla pagina iniziale
2
3 Dato apro RBlog
4 Allora posso visitare la pagina dell'autore
5 E posso visitare la pagina dell'abstract
```

Il primo scenario descrive la navigazione verso la homepage, relativo alla funzionalità 1.26 ed introduce la sintassi di Cucumber. Gli scenari sono caratterizzati da un titolo, che riassume il comportamento e da un insieme di passi. Seguendo la logica del BDD, ogni passo può alternativamente essere di tipo "*Given*", "*When*" e "*Then*".

Tramite i passi di tipo "Given" è possibile verificare che il sistema sia in uno stato prefissato e conosciuto all'utente, stato dal quale sarà possibile compiere le azioni descritte successivamente nel test. Rispetto ad uno use-case un passo "Given" è l'equivalente di una precondizione.

I passi "When" descrivono le azioni compiute e permettono la transizione del sistema verso un nuovo stato, analogamente agli eventi di una macchina a stati.

Lo scopo dei passi "Then" è di osservare il risultato delle azioni compiute precedentemente. Le osservazioni dovrebbero essere consistenti con i benefici dichiarati per la funzionalità ed analizzare solo quanto è osservabile tramite l'interfaccia del sistema ed ottenuto come conseguenza alle azioni compiute. Ad esempio, non è compito dei test di accettazione su RBlog verificare le tuple della tabella Post nel modello.

La scelta del prefisso del passo non limita le potenzialità del passo stesso, ma ovviamente un uso corretto favorisce la leggibilità del test. E' anche possibile sfruttare i prefissi "*And*" e "*But*" per rendere i propri scenari più scorrevoli; alle congiunzioni è attribuito il tipo del passo precedente.

Listato 1.29: Navigazione verso le pagine statiche

```
1 Schema dello scenario: Visita alla pagina dell'autore e alla pagina
      dell'abstract
2
3 Dato apro RBlog
4 Quando navigo verso "<nome della pagina>"
5 Allora la pagina è intitolata "<nome della pagina>"
```

```
6  E posso tornare alla pagina iniziale
7
8  Esempi:
9  | nome della pagina |
10 | Autore            |
11 | Abstract          |
```

Cucumber offre la possibilità di definire scenari parametrici. Sfruttando la sintassi *"Schema dello* scenario", la tabella "*Esempi*" e i delimitatori "< >" è possibile verificare tanti scenari quanti sono i parametri all'interno della tabella: lo scenario 1.29 è eseguito sia sulla pagina dell'autore che sulla pagina dell'abstract.

Listato 1.30: Testo generato dallo scenario 1.29

```
1  Scenario: Visita alle pagine statiche: la pagina dell'autore e all'
       abstract della tesi
2
3  Dato apro RBlog
4  Quando navigo verso "Autore"
5  Allora la pagina è intitolata "Autore"
6  E posso tornare alla pagina iniziale
```

Business Readable DSL Il linguaggio utilizzato per descrivere le funzionalità e gli scenari in Cucumber è Gherkin[40]. Gli autori, gli stessi di Cucumber, descrivono il linguaggio come Business Readable DSL[39], definizione introdotta da Martin Fowler nel 2008.

L'obbiettivo principale di un Business Readable DSL è permettere la partecipazione all'analisi e alla revisione del codice a figure non tecniche. Fowler sottolinea come sia più importante definire un linguaggio leggibile per tutte le diverse figure professionali coinvolte nello sviluppo rispetto ad uno anche scrivibile in maniera cooperativa. Un Business Readable DSL sopperisce alla necessità primaria di stabilire un canale di comunicazione arricchente fra le diverse parti che partecipano allo sviluppo, al contrario secondo Fowler, un Business Writeable DSL richiede un impegno troppo alto in termini di tempo e risorse umane coinvolte.

I18n Cucumber supporta attualmente 40 lingue, numero in rapida crescita secondo gli sviluppatori.[8] E' così possibile definire i passi degli scenari con le parole chiave nella lingua scelta: ad esempio per la definizione delle funzionalità di RBlog è stata utilizzata la lingua italiana. Per utilizzare uno dei pacchetti linguistici esistenti è necessario un header "# language: xx" all'interno dei file ".feature".

[8] Per ottenere la lista aggiornata delle lingue supportate è possibile eseguire il comando cucumber –i18n help o consultare la risorsa contenente il dizionario `https://github.com/cucumber/gherkin/blob/master/lib/gherkin/i18n.json`.

Supporto a Cucumber in RubyMine RubyMine, l'ambiente di sviluppo scelto per sviluppare tramite RoR, integra le funzionalità di Cucumber e assiste il programmatore nel corso dello sviluppo dei test: la sintassi di Gherkin è evidenziata, è presente l'auto-completamento delle parole chiave del DSL in tutte le lingue, è possibile navigare dalla definizione all'implementazione del passo ed è fornita un'interfaccia grafica per l'esecuzione dei test, configurabile in funzione delle etichette, file delle funzionalità da includere ed altri parametri.

1.3.2 Capybara

Capybara[41] è una libreria in Ruby che permette la definizione di test di accettazione automatici per applicazioni web, non necessariamente scritte tramite RoR, simulando le azioni eseguibili via interfaccia grafica.

Capybara nasconde all'utente i dettagli tecnici della navigazione tramite primitive di funzioni semplici, intuitive e versatili. Le funzionalità di Capybara sono astratte e mantengono la stessa prospettiva di un test effettuato manualmente da un utente.

La configurazione è effettuata tramite un breve script in Ruby: è necessario scegliere un browser web ed il relativo driver per Capybara importando la libreria nel file "*features/support/env.rb*".

Listato 1.31: Dipendenze all'interno dello script di configurazione.

```
require 'cucumber/rails'
require 'capybara'
require 'capybara/cucumber'
require 'capybara/rspec'
require 'capybara/poltergeist'
```

Infine, se necessario, è possibile configurare il driver scelto: la struttura di Capybara permette l'uso di diversi driver e browser. Tutti i driver[42] devono implementare le funzionalità obbligatorie indicate nella libreria, ma è consentito non fornire il resto delle operazioni ma essere comunque annoverati fra i driver esistenti per Capybara. Per RBlog è stato utilizzato PhantomJS[43] ed il driver Poltergeist[44], che verranno trattati in seguito.

Listato 1.32: Configurazione di Poltergeist

```
Capybara.default_driver = :poltergeist
Capybara.register_driver :poltergeist do |app|
  options = {
    :js_errors => true,
    :timeout => 120,
    :debug => true,
    :phantomjs_options => ['--load-images=yes', '--disk-cache=false
        '],
    :inspector => true,
  }
  Capybara::Poltergeist::Driver.new(app, options)
end
```

Le opzioni più rilevanti per Poltergeist sono:

- :js_errors rileva ogni errore relativo alle esecuzione di codice JavaScript e genera un errore in Ruby;
- :inspector è un'opzione sperimentale che permette il debug dell'esecuzione tramite una terza applicazione, come ad esempio Chrome web inspector[45];
- :debug reindirizza l'output dell'esecuzione in modalità debug verso STDERR.

PhantomJS

PhantomJS è un browser headless, supporta tutte le funzionalità di un browser web moderno ma senza possedere un'interfaccia grafica e si basa sul motore di rendering WebKit[46], lo stesso utilizzato da Chrome e Safari.

E' particolarmente adatto all'esecuzione automatica di applicazioni web non richiedendo di un framework per la gestione di GUI, funzionalità spesso mancante sui server che effettuano l'integrazione continua.

Con PhantomJS è anche possibile far coincidere il fallimento dei propri test in funzione di errori su codice JavaScript, funzionalità non presente su altri browser, grazie all'opzione :js_errors precedentemente descritta.[9]

Poltergeist

Poltergeist è un driver per il browser PhantomJS ed implementa la totalità delle funzionalità obbligatorie e molte delle opzionali disponibili. Dopo aver configurato il driver nel file "env.rb" tutte le operazioni saranno gestite tramite la libreria di Capybara.

La combinazione Capybara - Poltergeist - PhantomJS è attualmente una delle più diffuse per lo sviluppo di test di accettazione automatici in Ruby perché da ottimi risultati nella gestione di strumenti asincroni come AJAX[47], è estremamente veloce ed è molto accurata nella gestione di falsi positivi, come ad esempio il click su eventi esistenti nel DOM, Document Object Model, di una pagina HTML ma non visibili.[10] [11]

1.3.3 Implementazione dei passi

La struttura dei passi in Cucumber è definita a priori e non varia in funzione del tipo implementato. Per implementare un passo è necessario creare uno script Ruby, ad esempio "features/steps_definition/constraints.rb", e definirne l'implementazione.

[9] PhantomJS è installabile tramite i pacchetti d'installazione presenti sul sito ufficiale `http://phantomjs.org/download.html`.

[10] L'installazione di Poltergeist è effettuata attraverso il gemfile e la gemma "poltergeist".

[11] Quando viene richiesto il click su un elemento, Poltergeist non genera un evento attraverso il DOM ma simula un evento reale, ad esempio scendendo lungo la pagina nel caso in cui l'elemento non dovesse essere visibile. Inoltre sono previste diverse casistiche di errori, come l'impossibilità di compiere azioni su un elemento coperto.

Listato 1.33: Implementazione del passo "apro RBlog".

```
1 Given(/^apro RBlog$/) do
2   visit steps_helper.rblog_url
3   #...
4 end
```

Sfruttando il metodo Given di Cucumber, in generale lo schema è valido anche per gli altri tipi disponibili, è possibile specificare un'espressione regolare ed il comportamento associato. All'esecuzione dei test, è verificata la presenza un'implementazione per ogni passo definito. E' necessario che per ogni passo in Gherkin esista un'unica espressione regolare corrispondente fra i metodi disponibili.

Listato 1.34: Ipotetica implementazione ulteriore.

```
1 When(/^apro (.*)$/) do |site_name|
2   #...
3 end
```

L'introduzione di un'ulteriore implementazione, anche se con tipo differente, genere un'ambiguità che secondo il comportamento standard di Cucumber non è risolta.[12]L'implementazione di passo è definibile all'interno di un blocco e non esiste alcuna limitazione né controllo sulle espressioni che possono essere utilizzate.

L'implementazione dei passi non è vincolata dal tipo definito, ad esempio è possibile sfruttare un'asserzione come invariante. Questa particolarità di Cucumber offre la possibilità di definire più librerie di test di accettazione in differenti lingue ed utilizzare lo stesso codice per l'implementazione dei passi. Una potenzialità simile può essere utile per documentare lo sviluppo di progetti che coinvolgono stake-holders di diverse nazionalità.

La struttura di Capybara Vediamo quali funzionalità Capybara offra per semplificare la scrittura delle espressioni necessarie per l'implementazione dei passi in Cucumber.

Gli elementi di una pagina web sono indicati come nodi[48] in Capybara, la gerarchia è la seguente:

- la classe più semplice è Capybara::Node::Simple e rappresenta gli elementi di una pagina web, tali oggetti possono essere individuati all'interno del documento e analizzati in funzione degli attributi, ma non sono utilizzabili per compiere azioni;
- la classe Capybara::Node::Base è la classe padre di Capybara::Node::Element e Capybara::Node::Document, gli oggetti delle classi figlie condividono gli stessi metodi tramite i moduli Finders, Matchers e Actions. A differenza dei nodi semplici, sono utilizzabili per compiere azioni;

[12]Sfruttando l'opzione –guess di Cucumber è possibile far variare il comportamento per la scelta dell'implementazione da applicare.

- la classe Element rappresenta un singolo elemento all'interno del DOM della pagina;
- la classe Document rappresenta i documenti HTML nella loro interezza.

Le funzionalità di Capybara sono suddivise in tre moduli: "Finders", "Actions" e "Matchers"[49].

Il modulo "Finders" contiene un insieme di funzionalità dedicate all'individuazione di nodi all'interno della pagina. I metodi sono suddivisi in funzione del tipo di elemento ricercato, come ad esempio "find_button" e "find_link", e della cardinalità attesa: il metodo "all" restituisce tutti gli elementi che soddisfano la ricerca a differenza dei metodi "find_*" dai quali è atteso l'individuazione di esattamente un nodo.

Il modulo "Actions" permette l'interazione con l'interfaccia della pagina: sono quindi previsti, ad esempio, metodi per la compilazione di form HTML e la selezione di elementi. Le operazioni permettono anche di specificare delle opzioni per effettuare delle variazioni o verificare alcune proprietà prima di compiere l'evento.[13]

Infine il modulo Matchers verifica le proprietà di un nodo, sia esso un sotto elemento della pagina o il documento stesso. Ad esempio è possibile verificare che sia presente un selettore, indicando l'identificatore css o una query XPath, o la presenza di attributo per l'elemento che invoca il metodo.

La libreria di Capybara è ricca di funzionalità e si presta in maniera versatile a diversi usi e preferenze. Per la maggior parte dei metodi è prevista la possibilità di specificare delle opzioni ed influire, in funzione della natura dell'operazione, sul comportamento di default. Inoltre, soprattutto all'interno del modulo "Matchers", esistono diversi modi per definire le istruzioni, facilitando la scrittura dei test.

Navigare all'interno del sito Nel passo 1.33 è utilizzato il metodo "visit" che permette la navigazione verso una certa pagina web. All'esecuzione del metodo coincide una richiesta HTTP in GET all'indirizzo indicato come parametro, che può essere sia relativo che assoluto.[14]

Listato 1.35: Navigazione nel sito, sfruttando il testo visualizzato di un link.

```
1 When(/^navigo verso "([^"]*)"$/) do |page_name|
2   find_link(page_name).click
3 end
```

[13] Il metodo click_link permette di specificare l'opzione :href per verificare l'uguaglianza del attributo href prima di effettuare il click sul collegamento.

[14] Il metodo rblog_url dell'oggetto denominato steps_helper, appartiene alla classe StepsHelper, utilizzata per contenere alcuni metodi d'utilità sfruttati durante lo sviluppo dei test.

Il metodo "visit", utilizzato per aprire la pagina iniziale del blog, non verifica la presenza all'interno della pagina di un collegamento verso la destinazione, ma semplicemente effettua la richiesta all'indirizzo indicato.

Per verificare la presenza di link alle pagine statiche nell'homepage è stato utilizzato il metodo "find_link". Il metodo ricerca un collegamento all'interno della pagina in funzione dell'identificatore degli elementi HMTL "a" o del testo visualizzato. Come per le altre varianti dei metodi "find_*" in Finders, il metodo "find_link" solleva un'eccezione nel caso in cui la ricerca non dovesse individuare uno ed un solo elemento. Per navigare all'interno del sito web si invoca il metodo "click" sul nodo restituito, come mostrato nell'implementazione del passo 1.35.

A differenza del passo 1.33, dove non sono presenti parametri, nel passo 1.35 viene selezionato il collegamento che coincide con il valore di "page_name", tramite un blocco con un singolo parametro. Cucumber per ogni passo che contiene del testo fra virgolette, genera dei blocchi parametrici automaticamente. E' possibile applicare lo stesso procedimento manualmente, sostituendo all'interno dell'espressione del passo un proprio pattern e aggiungendo un parametro a cui attribuire il valore. I parametri all'interno dei passi hanno tipo stringa, ma è possibile definire delle conversioni di tipo tramite il metodo Transform[50].

Definizione delle asserzioni con RSpec All'interno della libreria di Capybara non sono presenti le funzionalità per la verifica di asserzioni, è quindi necessario sfruttare librerie terze, come ad esempio RSpec.

RSpec è un framework per il testing scritto in Ruby, le cui funzionalità sono suddivise in quattro moduli:

- RSpec-Core[51] fornisce la struttura per la definizione di funzionalità e scenari per il BDD;
- RSpec-Expectations[52] è una libreria di metodi per definire asserzioni;
- RSpec-Mocks[53] è un framework per l'implementazione di stub, oggetti mock, verifiche sull'invocazione di metodi e dell'interazione fra oggetti;
- RSpec-Rails[54] è un framework per la definizione di test sulle componenti che definiscono un'applicazione RoR, come il modello, i controlli e le viste ma anche gli Helper e l'instradamento delle richieste.

Oltre a RSpec è possibile l'integrazione all'interno di unit-test, Test::Unit[55] in Rails, oppure con le asserzioni definite in MiniTest::Spec[56].

La struttura di un'asserzione in RSpec è definita da due elementi: l'oggetto da verificare ed uno o più matcher, concatenati da operatori logici.[15]

[15] La maggior parte dei matcher di RSpec prevedono la verifica di una singola proprietà, ma esistono anche matcher compositi con arietà variabile. La lista completa è consultabile sulla documentazione ufficiale `https://www.relishapp.com/rspec/rspec-expectations/v/3-1/docs/composing-matchers`.

Dalla versione 2.11 di RSpec, la versione corrente è la 3.1.0, è stata modificata la sintassi del metodo "expect" per renderla più leggibile e versatile.

Listato 1.36: Esempi dell'uso del metodo "expect".

```
1 expect(obj).not_to <matcher>
2 expect{ ... }.to <matcher>
3 expect do
4 ...
5 ...
6 end.to <matcher>
7
```

Il metodo accetta un singolo parametro, sia esso un oggetto o un blocco, che viene verificato dai matcher indicati. Un matcher in RSpec è un metodo e fornisce un risultato booleano in funzione dell'operazione implementata e degli argomenti. L'uso corretto dei matcher è come parametri dei metodi "to" e "not_to".

Listato 1.37: Verifica la presenza del link

```
1 Then(/^posso visitare la pagina dell'autore$/) do
2   expect(find_link('Autore').visible?).to be_truthy
3 end
```

Lo scenario 1.29 richiede la possibilità di navigare verso la pagina statica dell'autore. L'asserzione è stata implementata sfruttando le funzionalità di Capybara, per individuare il collegamento in funzione del testo mostrato e ottenere la visibilità del nodo, e RSpec con il matcher "be_truthy".

Listato 1.38: Implementazione di un'asserzione parametrica.

```
1 Then(/^la pagina è intitolata "([^"]*)"$/) do |title_value|
2   expect(page.title).to eq(title_value)
3 end
```

Nello scenario 1.29 è richiesto il confronto di un parametro, definito attraverso l'espressione regolare, ed il titolo della pagina.[16] In RSpec esistono tre matcher per la verifica dell'uguaglianza: "equal?" verifica se le variabili si riferiscono allo stesso oggetto, "eql?" effettua un confronto sullo stato dell'istanza mentre l'operatore "==" confronta sia il tipo degli oggetti che i relativi stati, sfruttando eventuali conversioni.

Listato 1.39: Implementazione del passo "apro RBlog".

```
1 Given(/^apro RBlog$/) do
2   visit steps_helper.rblog_url
3   expect(page.status_code).to be == 200
4 end
```

[16] Capybara::DSL::page è un getter e restituisce la rappresentazione della pagina attualmente aperta nel browser.

Tramite il matcher "be" è possibile utilizzare gli operatori definiti in Ruby. Nella precondizione è verificato che lo stato HTTP sia equivalente a 200, che corrisponde alla corretta terminazione della richiesta.

1.4 Introduzione del CSS

Listato 1.40: Seconda funzionalità per RBlog

```
1 Feature: Introducendo il (S)CSS
2 Per rendere l'esperienza di navigazione gradevole
3 Come Lettore
4 Vorrei che il sito esponesse una grafica omogenea
```

Lo sviluppo attuale di RBlog prevede una semplice struttura e la navigazione fra le pagine esistenti, l'homepage e due pagine statiche contenenti una breve descrizione del progetto e della tesi. L'iterazione corrente introduce i fogli di stile e la verifica tramite Capybara degli effettivi cambiamenti nell'aspetto delle pagine.

Un aspetto importante nello sviluppo di applicazioni web è rispettare i principi di accessibilità consigliati dal W3C, il consorzio che si occupa della standardizzazione di internet e dei suoi servizi.

«The power of the Web is in its universality. Access by everyone regardless of disability is an essential aspect.» Tim Berners-Lee, W3C Director and inventor of the World Wide Web

Nella funzionalità non è richiesto che il sito sia accessibile da persone con disabilità, ma è comunque necessario verificare non solo la struttura delle pagine ma anche alcuni requisiti estetici siano rispettati: potrebbe essere necessario che le pagine verifichino un certo schema di colori o che sia presente un unico font.

Le caratteristiche estetiche dell'interfaccia e l'usabilità sono requisiti importanti nello sviluppo di un'applicazione web, ne consegue che anche gli strumenti per la definizione di test di accettazione dovrebbero offrire delle funzionalità per l'analisi di proprietà non esclusivamente legate alla struttura del DOM ed alle proprietà dei nodi, affinché tali richieste possano essere introdotte all'interno di scenari e funzionalità.

1.4.1 "CSS with superpowers"

RubyMine offre la possibilità di scegliere come implementare i propri fogli di stile: ovviamente è previsto l'uso del CSS3 per il quale è integrato il supporto, ma è anche disponibile sfruttare Sass[57], Scss e Less, estensioni dello standard che introducono nuove caratteristiche ai classici fogli di stile.

RubyMine genera per ogni controller un foglio di stile in Sass, e vista la completa integrazione di questo linguaggio nell'ambiente di sviluppo è stato deciso di sfruttarlo per il progetto.

Listato 1.41: Frammento del foglio di stile in Sass relativo ai post.

```
1 .posts {
2   #notice {
3     margin: 1em 15%;
```

```
4       text-align: center;
5       p {
6         color: forestgreen;
7       }
8     }
```

Sass fornisce alcuni strumenti sintattici per facilitare il riuso e l'organizzazione di regole: la caratteristica che maggiormente semplifica lo sviluppo dei fogli di stile è la possibilità di definire una gerarchia nelle regole. I fogli di stile appaiono più lineari e leggibili, rispecchiano la struttura del documento, è c'è un minor rischio di errori, non è necessario etichettare ogni elemento del DOM con identificatori e classi, ed è molto più semplice applicare delle correzioni a piccole porzioni dell'applicazione.

La sintassi di Sass prevede l'uso di variabili, ad esempio per salvare i riferimenti ad un colore o ad un font, consente l'importazione di altri documenti Sass e la definizione di fogli di stile parziali per fattorizzare alcuni elementi comuni dell'interfaccia. I file in Sass sono processati e compilati in un file CSS prima dell'utilizzo; il procedimento è gestito in automatico da RubyMine.

1.4.2 Testare il css

Per verificare le potenzialità di Capybara relativamente all'analisi "stilistica" delle pagine è stato scelto di analizzare il colore di sfondo di alcuni elementi. All'interno dell'intestazione delle pagine sono presenti i collegamenti introdotti dalla precedente funzionalità; nell'iterazione corrente sono stati definiti diversi fogli di stile che creano un semplice effetto cromatico: il colore dello sfondo dei collegamenti nell'intestazione cambia al passaggio del cursore.

Listato 1.42: Frammento del foglio di stile relativo ai collegamenti nell'intestazione di RBlog.

```
1 .banner_link:hover {
2   background-color: #8c2828;
3 }
```

Lo scenario relativo a questa funzionalità descrive il comportamento atteso.

Listato 1.43: Scenario relativo alla funzionalità.

```
1 Scenario: l'intestazione espone dei semplici effetti cromatici
2   Dato che è presente l'intestazione
3   E l'intestazione permette la navigazione
4   E i collegamenti non hanno sfondo
5   Quando il cursore si sposta sui collegamenti
6   Allora lo sfondo del collegamento cambia
```

Analisi delle proprietà dei nodi Il modulo Matchers di Capybara espone molti metodi e soluzioni equivalenti per la lettura dei valori presenti negli attributi dei nodi del documento, inoltre in HTML è possibile definire lo stile di un singolo nodo inserendo le regole desiderate all'interno dell'attributo "style".

Supponiamo che le pagine HTML dichiarino le regole di stile all'interno dei nodi stessi e non attraverso fogli di stile linkati nell'intestazione del documento HTML: durante l'analisi del valore di un attributo strutturato come "style" Capybara non effettua alcuna operazione straordinaria. Viene quindi estratta una stringa da analizzare tramite espressioni regolari create ad hoc per la regola CSS a cui si è interessati.

Analisi dei fogli di stile e corrispondenza all'interno del DOM La pratica di definire lo stile direttamente all'interno delle pagine HTML è deprecata in quanto minimizza il riuso del codice e rende estremamente fragile l'insieme delle viste in termini di manutenibilità.

Capybara, più precisamente Poltergeist, non effettua alcuna valutazione dei fogli di stile allegati alla pagina e non associa le regole presenti ai rispettivi nodi del DOM. Le regole sono ignorate e l'accesso agli attributi "style" o "background-color" restituiscono valore nullo.

Listato 1.44: Asserzione fallita sull'attributo "style" per i collegamenti dell'intestazione.

```
1 expect(banner_link_div[:style]).not_to be_nil
```

Preso atto delle mancanze di Capybara e Poltergeist, per completare l'implementazione dello scenario è stato individuata una soluzione alternativa tramite la funzionalità del framework di eseguire script JQuery all'interno della pagina.

JQuery è una libreria scritta in JavaScript che permette l'analisi e manipolazione del DOM, gestisce gli eventi all'interno della pagina, le animazioni e fornisce delle interfacce per semplificare l'uso di Ajax.

Listato 1.45: Analisi del colore di sfondo dell'elemento via JQuery.

```
1 def background_color(id, page)
2   jscript = "$('#{id}').css('backgroundColor')"
3   page.evaluate_script(jscript)
4 end
```

Il metodo utilizza il selettore generato dall'identificatore e accede alla proprietà desiderata, il metodo "css" di JQuery supporta tutte le funzionalità del CSS3, permettendo sia la lettura che la modifica degli elementi.

Purtroppo però, anche questa soluzione è parziale. La proprietà principale delle regole del CSS è l'ereditarietà: la definizione di una regola per un certo elemento ha conseguenze anche sui sotto elementi presenti nel DOM. Gli attributi stilistici si propagano a cascata, se applicabili, e hanno conseguenze in funzione dell'importanza, la regola può essere definita dal browser web, dall'utente o dall'autore del sito, e dalla specificità della definizione.

Listato 1.46: Precondizione sul colore dei collegamenti nell'intestazione.

```
1 Given(/^i collegamenti non hanno sfondo$/) do
```

```
2     #header_rgb_background = steps_helper.background_color(
          steps_helper.header_id, page)
3     @textual_header_link_divs.each do |banner_link_div|
4       id = banner_link_div[:id]
5       background_color = steps_helper.background_color("##{id}", page
            )
6       expect(background_color).to eq('rgba(0, 0, 0, 0)')
7       #expect(steps_helper.background_color("##{id}", page)).to be
            eq(header_rgb_background)
8     end
9   end
```

Nell'implementazione del passo è presente un'asserzione commentata che impedisce il successo del test. JQuery infatti non effettua alcuna verifica sull'elemento "div" corrispondente all'intestazione e per il quale è presente un colore di sfondo, restituendo il valore di default "rgba(0, 0, 0, 0)".

Sia lo scenario che è stato descritto che quelli non trattati della funzionalità 1.40 non verificano proprietà elaborate dei fogli di stile, ma le limitazioni presenti non permettono di sfruttare i test di accettazione per la verifica dei principi di accessibilità né per aspetti più elementari della grafica delle pagine web.

1.4.3 Il contesto degli scenari

In Cucumber è possibile dichiarare all'interno delle funzionalità un contesto, definito da un numero arbitrario di passi.

Listato 1.47: Background della funzionalità 1.40.

```
1 Contesto:
2   Dato apro RBlog
```

I passi definiti nel contesto sono eseguiti prima di ogni scenario appartenente alla funzionalità e sono utilizzati per fattorizzare alcune premesse e per rendere più incisivi gli scenari. Per evitare di complicare eccessivamente i test di accettazione è consigliato utilizzare un numero di passi ridotto nel contesto per mantenere alta la leggibilità degli scenari.[17]

Nell'esempio è stato definito un contesto evitare la ripetizione del passo relativo alla prima iterazione in tutti gli scenari della funzionalità.

1.4.4 Debug con Capybara

Utilizzare Poltergeist e PhantomJS semplifica la verifica di applicazioni web che sfruttano JavaScript e metodi asincroni, ma non offre all'utente la possibilità di verificare tramite la GUI, Graphical User Interface, le azioni che vengono effettuate nei test.

Per effettuare il debug, Poltergeist offre alcuni metodi per catturare la pagina corrente: tramite il metodo "save_and_open_page" si ottiene il codice HTML

[17] Una pratica empirica suggerisce di mantenere le funzionalità essenziali per permettere la lettura del contesto e dello scenario senza dover scorrere nella schermata.

del DOM, mentre con il metodo "save_and_open_screenshot" viene catturata e salvata la schermata del browser.

All'interno dell'implementazione dei passi è possibile introdurre dei breackpoint, ma l'interfaccia e la proprietà che i nodi espongono non permette una semplice analisi. Durante lo sviluppo è stata combinata la possibilità di salvare le schermate del browser e inserire delle interruzioni nell'esecuzione dei test per verificare manualmente lo stato dell'esecuzione.

1.4.5 XPath

Le funzionalità del modulo Finders permettono di effettuare query tramite XPath[58].

Listato 1.48: Query per la selezione del primo elemento del "body".

```
1 Then(/^l'intestazione è posizionata all'inizio$/) do
2   header = page.find(:xpath, 'descendant::body/*[1]')
3   expect(@header).to eq(header)
4 end
```

Il vantaggio di creare dei selettori attraverso query XPath consiste nella possibilità di introdurre sia vincoli sulla struttura, nel frammento di codice è selezionato il primo elemento appartenente al "body" della pagina, sia vincoli sul contenuto degli attributi e dei valori.

La definizione di selettori tramite CSS permette la definizione di query sulla struttura del DOM, ma è possibile solo verificare il testo dei nodi.

Listato 1.49: Query per la selezione dei "div" con un titolo corrispondente al paramentro.

```
1 def post_divs_matching_title(page, post_title)
2   xpath_query = "//div[@class = 'post'][p/a[contains(text(),'#{
        post_title}')]]"
3   page.all(:xpath, xpath_query)
4 end
```

Il metodo "find" di Capybara, ma in generale tutti i metodi del modulo Finders, supportano la definizione di selettori attraverso XPath e CSS ma non è distinto automaticamente il tipo di espressione utilizzata: nei frammenti di codice i metodi sono stati invocati con il simbolo ":xpath".

1.5 Definizione del modello

Listato 1.50: Funzionalità dell'iterazione.

```
1 Funzionalità: Gestione dei post
2   Come Autore
3   Vorrei poter inserire, leggere, modificare e rimuovere dei post
        su RBlog
4   Per poter documentare la tesi
```

L'obbiettivo dell'iterazione corrente è aggiungere delle pagine dinamiche per la gestione dei post su RBlog. Le funzionalità supportate sono le CRUD, Create, Read, Update, Delete. La generazione dei controller e del modello è stata fatta sfruttando le funzionalità a riga di comando del comando "rails generate" che permette la definizione di tutte le componenti presenti nel framework attraverso una sintassi intuitiva.

Le componenti generate richiedono ovviamente di essere personalizzate ma espongono una struttura completa che minimizza la configurazione.

1.5.1 Dipendenze

Nella definizione degli scenari è conveniente utilizzare uno stile che favorisca sia lo sviluppatore, che ha il compito di scrivere e mantenere i test di accettazione, sia gli stakes holder, che tramite gli scenari possono seguire lo sviluppo del prodotto.

Gli scenari devono essere indipendenti fra loro e l'ordine di esecuzione non deve aver conseguenze sul risultato. Mantenere delle dipendenze funzionali all'interno nella libreria di testing può introdurre diverse problematiche all'aumentare della cardinalità dei test e della complessità o alla variazione dello condizioni d'esecuzione, introducendo ad esempio l'esecuzione in parallelo di più scenari.

Gli scenari della funzionalità 1.50 sfruttando le interfacce di RBlog per creare, modificare e rimuovere post. E' quindi necessario introdurre delle procedure per annullare le modifiche compiute.

Hooks Cucumber definisce degli istanti durante l'esecuzione dei test ai quali "agganciare" l'esecuzione di eventi definiti dallo sviluppatore.

Listato 1.51: Creazione di un nuovo post.

```
1 Scenario: Scrittura di un nuovo post
2   Dato il post "Lorem Ipsum" non è leggibile su RBlog
3   E apro la pagina per la creazione di un nuovo post
4   Quando inserisco "Lorem Ipsum" come titolo
5   E inserisco del testo riempitivo come contenuto
6   E salvo il post
7   Allora il post "Lorem Ipsum" è stato creato con successo
8   E il post "Lorem Ipsum" è leggibile su RBlog
```

Lo scenario si conclude con la creazione di un nuovo post dal titolo "Lorem Ipsum", eseguendo nuovamente il test la prima pre-condizione non sarebbe verificata. Tramite il meccanismo degli hook[59] in Cucumber sono state definite delle procedure per annullare tutte le modifiche effettuate sull'applicazione.[18]

Listato 1.52: Hook eseguito al termine degli scenari.

```
1 After('@clear') do
2   clear_all_ipsums(page)
3 end
```

Nel frammento è utilizzato il metodo "After", a cui è attribuita l'operazione da eseguire al termine dello scenario. Il metodo è applicato esclusivamente agli scenari che sono etichettati con "@clear", se fosse utilizzato senza parametri l'esecuzione avverrebbe al termine di ogni scenario della libreria.

Il metodo "clear_all_ipsums" verifica la presenza di articoli i cui titoli contengano il testo "Lorem Ipsum" e ne effettuano la cancellazione.

Il metodo "After", ma in generale anche "Before" e gli hook per i singoli passi "AfterStep" e "BeforeStep", sono utilizzabili con un numero arbitrario di etichette. Inoltre è possibile definire dei blocchi con un singolo parametro rappresentante lo scenario che espone delle funzionalità per verificare il risultato dello scenario.

```
1 Around('@fast') do |scenario, block|
2   Timeout.timeout(0.5) do
3     block.call
4   end
5 end
```

Il metodo "Around" invece è utilizzato per essere eseguito "intorno" allo scenario e poter effetture delle misurazioni sulla velocità degli scenario. Lo scenario è passato al blocco del metodo attraverso un ulteriore parametro.

Tool disponibili Per non introdurre un alto numero di librerie all'interno del progetto, gli hook eseguono le azioni di regressione attraverso l'interfaccia grafica dell'applicazione utilizzando le stesse funzionalità presenti nei passi.

Esistono però delle librerie che, attraverso una sintassi semplificata, offrono le funzionalità per la creazione e manipolazione di nuovi elementi all'interno del modello come FactoryGirl, o strumenti come DatabaseCleaner che offrono le funzionalità per l'eliminazione di quanto presente nel sistema attraverso diverse strategie.

Nel caso di librerie di test complesse potrebbe essere conveniente utilizzare uno di questi strumenti per semplificare lo sviluppo e concentrarsi esclusivamente sullo verifica degli scenari.

[18] La definizione degli hook non richiede alcuna configurazione, è sufficiente inserire il nuovo file all'interno della cartella contenente l'implementazione dei passi.

1.5.2 Gestione dei form

Per le operazioni di inserimento e modifica dei post in RBlog sono state definite due viste e un semplice form, contenuto all'interno di una vista parziale. La gestione degli eventi da parte di Capybara e Poltergeist è uno degli aspetti in cui la coppia framework - driver eccelle.

```
1 When(/^inserisco "([^"]*)" come titolo$/) do |title_value|
2   page.fill_in 'post_title', :with => title_value
3 end
```

I campi dei form sono compilati attraverso il metodo "fill_in" che individua i nodi HTML "input" o "text_area" in funzione del nome del campo e inserisce il testo specificato dal parametro ":with".

```
1 When(/^salvo il post$/) do
2   click_button 'submit'
3 end
```

Le funzionalità del modulo Actions sono estremamente semplici ed intuitive: sono supportati diversi tipologie di campi, dalle aree testuali ai check box, è possibile selezionare un file da allegare tramite il metodo "attach_file". Tutti i metodi presenti offrono allo sviluppatore del driver di estendere le funzionalità attraverso un parametro "options" di tipo Hash, l'implementazione della struttura dato dizionario in Ruby.

Il metodo within Per semplificare l'esecuzione di più operazioni che condividono lo stesso nodo è disponbile il metodo "within", che esegue il blocco associato nel contesto del nodo passato come parametro.

```
1 When(/^cancello il post "([^"]*)"$/) do |post_title|
2   expect(page.has_content?(post_title)).to be_truthy
3   post_div = steps_helper.post_div_by_title(page, post_title)
4   within(post_div) do
5     #...
6     find('.remove_post_button').click
7   end
8 end
```

1.6 Login & Autorizzazione

L'obbiettivo della funzionalità è l'introduzione di un meccanismo di autenticazione e gestione delle autorizzazioni in RBlog.

Listato 1.53: Descrizione della funzionalità di autenticazione.

```
1 Funzionalità: Autenticazione su RBlog
2   Come Autore di RBlog
3   Vorrei che alcune operazioni sensibili siano permesse previa
        autenticazione
4   Per poter garantire l'autenticità dei contenuti
```

Per supportare le operazioni di login e logout all'interno del blog è stata modificata l'applicazione in tutte le sue componenti. All'interno del modello è stato introdotta l'entità autore, definita da un indirizzo email, unico all'interno del dominio, una password ed una relazione uno a molti con i post.

Per rendere più realistico il meccanismo di autenticazione, non è mantenuta la password in chiaro ma la coppia impronta hash e salt, riducendo la sensibilità ad attacchi di tipo dizionario sull'impronta hash della password. Il salt è una stringa casuale da concatenare alla password in chiaro per generare impronte hash più sicure. La generazione del sale e il calcolo delle impronte è effettuato tramite la gemma BCrypt[60].

Listato 1.54: Frammento del controllo per l'autenticazione.

```
1  class SessionsController < ApplicationController
2    def create
3      author = Author.authenticate(params[:email], params[:hpassword
           ])
4      if author
5        session[:author_id] = author.id
6        redirect_to
7          :root, :notice => 'Login effettuato, benvenuto!'
8      else
9        redirect_to :log_in, :notice => 'Credenziali invalide.'
10     end
11   end
```

Il controllo "SessionController" è stato aggiunto all'applicazione, dichiarando l'instradamento per le richieste alla pagina di login, la creazione della sessione ed il logout. Il login su RBlog coincide con la creazione di una sessione con attributo l'identificatore univoco dell'autore. Rails fornisce le interfacce per la gestione di sessioni, che vengono criptate di default.

1.6.1 Black-box Testing

Lo sviluppo di test di accettazione automatici prevede di considerare il sistema da una prospettiva esterna, senza alcuna conoscenza dell'implementazione sottostante. I test sono definiti e verificati osservando il comportamento di una

scatola nera e non dovrebbero fare assunzioni sul comportamento del software.

Nonostante questo requisito, i test sull'autenticazione rilassano il principio del black-box testing per verificare le potenzialità di Capybara nella gestione di sessioni e cookie.

Listato 1.55: Accesso alla sessione.

```
1 def encrypted_session(page)
2   cookies = page.driver.cookies
3   cookies.values[0].value
4 end
```

E' importante ricordare che l'architettura di Capybara prevede l'utilizzo di driver e funzionalità a basso livello, come la gestione delle sessioni, potrebbero variare in funzione delle componenti scelte.

Poltergeist implementa le funzionalità per la lettura dei cookie, come mostrato nel metodo "encrypted_session", definendo diversi metodi per accedere alle proprietà.[19]

Inoltre è possibile creare e rimuovere i cookie, funzionalità utili nella definizione degli hook di Cucumber.

Listato 1.56: Login su RBlog.

```
1  When(/^mi autentico come "([^"]*)"$/) do |email|
2    pre_login_encrypted_session = steps_helper.encrypted_session(page
         )
3    visit steps_helper.login_page_url
4    /*.....*/
5    /*Login*/
6    /*.....*/
7    post_login_encrypted_session = steps_helper.encrypted_session(
         page)
8    expect(pre_login_encrypted_session).not_to
9    eq(post_login_encrypted_session)
10 end
```

L'implementazione del passo di login comprende la compilazione dei campi relativi all'email e alla password ed una semplice asserzione sul valore criptato della sessione: dopo la verifica delle credenziali viene aggiunto un attributo facendo variare la codifica.

1.6.2 Manutenibilità

L'introduzione dell'autenticazione è l'unica funzionalità che abbia inciso su tutte le componenti dell'applicazione ed anche sui test di accettazione già presenti.

Listato 1.57: Variazioni nella funzionalità di creazione dei post.

```
1 @cap3
2 @clear_and_logout
3 Funzionalità: Gestione dei post
```

[19] Sono presenti i metodi: "name", "value", "domain", "path", "secure?", "httponly?", "expires".

```
4     Come Autore
5     Vorrei poter inserire, modificare e rimuovere dei post su RBlog
6     Per poter documentare la mia tesi
7
8     Contesto:
9     Dato che apro RBlog
10    E mi autentico come "mattia@rblog.io"
```

Introducendo la verifica delle autorizzazioni, l'accesso alle funzionalità di creazione, modifica e cancellazione di un post non sono più eseguibile senza aver compiuto l'autenticazione.[20] E' stato quindi necessario modificare il contesto di alcune funzionalità specificando l'azione di login ed estendere le operazioni di regressione dopo ogni scenario per includere il logout.

Listato 1.58: Il nuovo hook combina regressione del modello e logout.

```
1  After('@clear_and_logout') do
2    clear_all_ipsums(page)
3    logout(page)
4  end
```

La precedente etichetta "@clear" è stata sostituita da "@clear_and_logout", non è stata trovato alcun riferimento sull'ordine di esecuzione degli hook che avrebbe permesso l'introduzione di una nuova etichetta piuttosto che la modifica di quella esistente.

Le variazioni riguardanti le operazioni di regressione hanno ulteriormente complicato la funzionalità: ritengo che l'uso di etichette, sintatticamente più vicine a Java che al linguaggio naturale, stoni all'interno di un documento scritto attraverso un linguaggio Business Readable come Gherkin.

L'uso di etichette su scenari e funzionalità dovrebbe essere utile solo per organizzare i test in maniera non funzionale, come categorizzare in base al tempo di esecuzione richiesto -ad esempio @rapido, @standard e @lento- o stabilendo i giusti intervalli di l'esecuzione su un server per la CI -ad esempio @sempre, @ogni_ora, @ogni_notte-.

[20] Dalle pagine sono rimossi i collegamenti alle azioni riservate e l'accesso diretto alle pagine senza aver effettuato l'autenticazione causa la ridirezione verso la pagina di login.

1.7 Asincronia

L'obbiettivo delle prossime funzionalità è verificare le potenzialità di Capybara con Javascript, JQuery, JQuery UI e Ajax, strumenti che permettono la definizione di comportamenti asincroni.

1.7.1 JavaScript

Listato 1.59: Introduzione di un breve script Javascript.

```
Funzionalità: Easter Egging
  Come Sviluppatore
  Vorrei che nel blog fosse presente un mio logo
  Per firmare il mio lavoro
```

Tramite questa funzionalità è introdotto un piccolo script Javascript, associato all'evento "onclick" del "div" piè di pagina.

Listato 1.60: Footer di RBlog.

```
<div id="footer" onclick="switch_easter_egg()">
  <p>@ 2014 - Mattia</p>
</div>
```

Listato 1.61: Frammento della funzione per aggiunta e rimozione del logo.

```
function switch_easter_egg() {
  var woodstock = $('#woodstock');
  if (!woodstock.length) {
    var img = document.createElement("img");
    img.src = "/assets/woodstock.png";
    img.id = "woodstock";
    /*...*/
    document.getElementById("footer").appendChild(img);
  }else{
    woodstock.remove();
  }
}
```

La funzione, tramite un selettore JQuery, aggiunge o rimuove un piccolo logo in fondo alla pagina. Il click sull'elemento "#footer" esegue la funzione e modifica di conseguenza il DOM.

Capybara è stato sviluppato sotto l'assunzione che nello sviluppo di applicazioni web moderne potenzialmente ogni elemento potrebbe essere il risultato di un comportamento asincrono; per ogni operazione sul DOM è concesso che l'elemento debba ancora apparire. Capybara permette la configurazione di un parametro di attesa che specifica il tempo massimo, dopo il quale verrà sollevata un'eccezione.

```
Given(/^non è presente il logo nell'intestazione$/) do
  step 'è presente il pié di pagina'
  expect(@footer.has_css?('img')).to be_falsy
```

```
4     expect(@footer.has_css?('#woodstock')).to be_falsy
5   end
```

L'implementazione dei passi non subisce alcuna modifica in funzione del tipo di comportamento dell'elemento sotto test. Non è quindi necessario specificare manualmente dei timeout o delle pause arbitrarie nei test, ma è Capybara stesso a gestire quest'aspetto.

1.7.2 JQueryUI

JQuery UI[61] è una libreria grafica per l'introduzione di plugin grafici all'interno di applicazioni web. Attualmente alla versione 1.11, è sviluppata sfruttando JQuery e permette una rapida integrazione di widget, come menu con auto-completamento o selettori di date, ed effetti grafici. Le uniche dipendenze richieste sono JQuery e Javascript.

All'interno di RBlog, ed in particolare nell'intestazione del sito, è stato introdotto un semplice form HTML per ricercare i post in funzione del titolo. La ricerca definisce un parametro HTTP che è analizzato dallo stesso controllo che popola la home page.

Il comportamento tipico dei widget definiti in JQuery UI è intuitivo: le azioni eseguite in maniera asincrona modificano il DOM della pagina corrente, aggiungendo o modificando un insieme di nodi. Per ogni plugin è presente un foglio di stile che descrive l'aspetto grafico dell'elemento introdotto. Ad esempio il menu con auto-completamento aggiunge una lista in HTML, attraverso i nodi "ul" e "il", come ultimi elementi del "body". La lista numerata e gli elementi sono però visualizzati immediatamente al di sotto del campo "input" a cui si riferiscono grazie ai fogli di stile.[21]

Ajax I parametri che definiscono il widget del menu permettono di indicare quale sia la sorgente per auto-completare il testo immesso nel campo di "input". E' possibile dichiarare staticamente i dati oppure fornirli in maniera dinamica in funzione del testo inserito.

Listato 1.62: Funzione di autocompletamento in JavaScript.

```
1  function autocomplete() {
2    if ($("#search_input_text").length) {
3      $("#search_input_text").autocomplete({
4        source: function (request, response){
5          $.ajax({
6            url: "/posts/autocomplete_title",
7            data: {title: $("#search_input_text").val()},
8            success: function (data) {
9                   response(data);
10           },
```

[21] JQuery UI definisce anche diversi temi per i plugin, garantendo una migliore integrazione estetica.

```
11            failure: function () {
12                    console.log("Failure");
13            }
14          })
15        },
16        minLength: 2,
17        focus: function (event, ui) {
18          $("#search_input_text").val(ui.item.value);
19        },
20        select: function (event, ui) {
21          $("#search_input_text").val(ui.item.value);
22        }
23      });
24    }
25  }
```

AJAX, acronimo di Asynchronous JavaScript and XML, è una libreria in JavaScript per lo scambio di dati fra il web-browser ed il server che ospita l'applicazione web. Il comportamento è definito asincrono in quanto le informazioni che sono ottenute tramite la libreria, sono caricati in background senza interferire con il comportamento della pagina.

In RBlog la compilazione del campo per la ricerca di post effettua una chiamata AJAX ad un controllo che, ricevuto un parametro GET, restituisce un array di stringhe serializzate tramite Json contenente i titoli dei post per completare la ricerca.
La funzione "autocomplete" è associata alla pagina di RBlog tramite gli eventi di JQuery.

Listato 1.63: Callback per la funzione "autocomplete".

```
1 $(window).bind('page:change', autocomplete);
2 $(document).ready(autocomplete);
```

1.7.3 Scenari sull'auto-completamento

Rispetto alla funzionalità 1.59, dove il DOM è modificato tramite una chiamata Javascript in maniera praticamente istantanea, introdurre l'auto-completamento della ricerca introduce l'utilizzo di AJAX ed un maggior ritardo nell'aggiornamento della pagina.

Listato 1.64: Introduzione della ricerca.

```
1 Funzionalità: Ricerca fra i post
2   Come Lettore
3   Vorrei poter ricercare i post su RBlog
4   Per poter navigare fra i contenuti più velocemente
```

L'obbiettivo della funzionalità è verificare le potenzialità di Capybara nella gestione di chiamate AJAX e nell'utilizzo di widget di JQuery UI.

Fino alla versione 2.0 Capybara includeva il metodo "wait_until" per la verifica di elementi potenzialmenti asincroni. Il metodo nell'ultimo rilascio della libreria è stato integrato in tutte le funzionalità e rimosso, permettendo di definire test senza la necessità di definire timeout e attese.

E' ora presente una nuova funzionalità per verificare pagine web che espongono comportamenti asincroni. Il metodo "synchronize" esegue il blocco associato fino a che non ha successo, permettendo il corretto completamento delle funzioni AJAX. L'esecuzione del blocco dipende dal tempo di attesa definito in Capybara, configurabile in funzione delle necessità.

Listato 1.65: Utilizzo del metodo "syncronize".

```
def finished_all_ajax_requests?(page)
  page.document.synchronize do
    page.find('#ui-id-1')
  end
end

def wait_for_ajax(page)
  begin
    Timeout.timeout(Capybara.default_wait_time)
    do
      loop until finished_all_ajax_requests?(page)
    end
  rescue Timeout::Error
  end
  yield if block_given?
end
```

Il vantaggio di utilizzare il metodo "synchronize" rispetto all'utilizzo classico della libreria, che non prevede l'introduzione di attese o polling, consiste nella verifica delle situazioni dove un elemento è presente sulla pagina, come la lista numerata del plugin di JQuery UI, ma è necessario dare il tempo al browser di completarne il rendering, in quanto le funzionalità di Capybara sono più veloci. Lo svantaggio dell'utilizzo del metodo "synchronize" consiste nell'introduzione di stalli per l'esecuzione di test in cui gli elementi asincroni non debbano apparire, come una ricerca senza suggerimenti. Inoltre, l'attesa per un elemento potrebbe mascherare componenti lente all'interno del sito e un cattivo Look and Feel, richiedendo comunque una verifica manuale dell'applicazione web.

Le difficoltà nella verifica di widget asincroni sono causate sia dalla necessità di prevedere ritardi nella visualizzazione sia nella simulazione dell'interazione con le componenti.

```
Scenario: Autocompletamento della ricerca
  Dato nell'intestazione è presente la barra di ricerca
  Dato il post "Lorem Ipsum" esiste
  Quando inserisco il testo "lor" da ricercare
  Allora viene proposto il post "Lorem Ipsum"
  Quando inserisco il testo "xyz" da ricercare
  Allora non è proposto alcun post
  ...
```

L'auto-completamento in RBlog prevede che la compilazione del campo di ricerca fornisca alcuni suggerimenti all'utente. Inoltre al passaggio del cursore su una delle voci suggerite o utilizzando la tastiera per selezionare un'opzione, il testo digitato dall'utente è sostituito dalla voce corrente.

Queste interazioni con il widget prevedono la definizione di passi che esplicitano una successione di eventi sugli elementi del DOM.

Listato 1.66: Compilazione del menù con autocompletamento.

```
When(/^inserisco il testo "([^"]*)" da ricercare$/) do |
    searched_text|
  page.fill_in 'search', :with => searched_text
  search_input = page.find('#search_input_text')
  search_input.trigger(:focus)
  #search_input.trigger(:keydown)
  page.execute_script %Q{ $('#search_input_text').trigger('keydown
      ') }
  steps_helper.wait_for_ajax page
end
```

Nel passo è inserito il valore del parametro "searched_text", eseguito il focus dell'elemento "input" tramite il quale effettuare la ricerca ed è invocata la pressione del tasto "freccia giù" per selezionare il primo elemento. Il passo si conclude con l'attesa che il menù sia completamente renderizzato.

Poltergeist non implementa tutti i possibili eventi disponibili, è quindi necessario sfruttare script JQuery per colmare le mancanze del driver.

Listato 1.67: Ricerca di un post tramite click di una delle opzioni proposte.

```
When(/^ricerco "([^"]*)"$/) do |searched_text|
  step "inserisco il testo \"#{searched_text}\" da ricercare"
  steps_helper.wait_for_ajax page do
    regexp = Regexp.new(Regexp.escape(searched_text), 'i')
    if page.has_css?('.ui-menu-item', :text => regexp)
      post_hint = page.find(:xpath, "//li[@class = 'ui-menu-item']"
          )
      expect(post_hint.text).to match(%r{#{searched_text}}i)
      post_hint.trigger(:mouseenter)
      post_hint.click
    end
    find('#search_icon').click
  end
end
```

Rispetto al passo precedente, dove il testo da ricercare veniva esclusivamente inserito, nell'implementazione del evento corrente viene completata la ricerca.

Invocando il passo precedente il form è compilato, mentre le azioni sono eseguite all'interno del blocco fornito al metodo "wait_for_ajax".

Con il metodo "has_css", che individua nel DOM l'elemento che coincide con selettore ed il cui testo soddisfa l'espressione regolare, si ottiene un riferimento al nodo sul quale spostare il cursore ed effettuare il click. Per completare la ricerca è eseguito un ulteriore click sull'immagine che effettua l'invio del form.

La difficoltà nell'utilizzo di plugin definiti tramite librerie grafiche come JQuery UI o Bootstrap, consiste nella razionalizzazione degli eventi che manualmente vengono compiuti per utilizzare le interfacce grafiche.

```
1 Then(/^viene proposto il post "([^"]*)"$/) do |hint|
2   ui_menu_items = page.all(:xpath, "//li[@class = 'ui-menu-item'][
        text() = \"#{hint}\"]")
3   expect(ui_menu_items.length).to be == 1 end
```

Avendo gestito l'asincronia ed utilizzato i giusti eventi, le asserzioni non presentano ulteriori accorgimenti.

Acronimi

Di seguito sono elencati gli acronimi utilizzati all'interno della tesi, in ordine alfabetico.

ATDD: Acceptance Test-Driven Development

BDD: Behavior-Driven Development

CRUD: Create, Read, Update, Delete

CSS: Cascading Style Sheets

CVS: Concurrent Versions System

DOM: Document Object Model

ERB: Embedded RuBy

GUI: Graphical User Interface

HTML: HyperText Markup Language

HTTP: HyperText Transfer Protocol

IDE: Integrated Development Environment

MVC: Model-View-Controller

REST: REpresentational State Transfer

RoR: Ruby on Rails

SSH: Secure SHell

TDD: Test-Driven Development

URL: Uniform Resource Locator

Bibliografia

[1] https://www.ruby-lang.org/it/ 1

[2] http://ruby-doc.org/core-2.1.4/Symbol.html 2

[3] http://www.ruby-doc.org/core-2.1.4/String.html 2

[4] https://www.ruby-lang.org/en/news/2014/09/18/ruby-2-2-0-preview1-released/ 3

[5] http://www.ruby-doc.org/core-2.1.4/Hash.html 3

[6] http://www.scribd.com/doc/27174770/Garbage-Collection-and-the-Ruby-Heap 3

[7] http://www.ruby-doc.org/core-2.1.4/Object.html 3

[8] http://www.ruby-doc.org/core-2.1.4/BasicObject.html 3

[9] http://rubykoans.com/ 5

[10] https://rubymonk.com/ 5

[11] https://www.jetbrains.com/ruby/ 5

[12] http://www.martinfowler.com/eaaCatalog/activeRecord.html 6

[13] http://guides.rubyonrails.org/action_controller_overview.html 7

[14] http://api.rubyonrails.org/classes/ActionController/Base.html 7

[15] http://guides.rubyonrails.org/routing.html 7

[16] http://guides.rubyonrails.org/action_controller_overview.html#filters 8

[17] http://en.wikipedia.org/wiki/ERuby

[18] http://guides.rubyonrails.org/layouts_and_rendering.html#understanding-yield 9

[19] http://en.wikipedia.org/wiki/ERuby

[20] http://api.rubyonrails.org/classes/ActionView/Helpers/UrlHelper.html 9

[21] http://guides.rubyonrails.org/testing.html 10

[22] https://github.com/rspec/rspec 10

[23] https://github.com/seattlerb/minitest 10

[24] https://github.com/DatabaseCleaner/database_cleaner 10

[25] https://github.com/thoughtbot/factory_girl 10

[26] http://guides.rubyonrails.org/testing.html##rake-tasks-for-running-your-tests 10

[27] https://rubygems.org/ 10

[28] http://maven.apache.org/ 10

[29] http://www.gradle.org/ 10

[30] http://bundler.io/gemfile.html 10

[31] https://github.com/rails/spring 11

[32] http://git-scm.com/docs/git-revert 11

[33] http://guides.rubyonrails.org/migrations.html 11

[34] http://api.rubyonrails.org/classes/ActionView/PartialRenderer.html 12

[35] http://www.yaml.org/ 13

[36] http://www.postgresql.org/ 14

[37] http://cukes.info/ 15

[38] http://www.agilemodeling.com/artifacts/userStory.htm 15

[39] 17

[40] https://github.com/cucumber/cucumber/wiki/Gherkin 17

[41] http://jnicklas.github.io/capybara/ 18

[42] https://github.com/jnicklas/capybara#drivers 18

[43] http://phantomjs.org/ 18

[44] https://github.com/teampoltergeist/poltergeist 18

[45] https://developer.chrome.com/devtools 19

[46] https://www.webkit.org/ 19

[47] http://en.wikipedia.org/wiki/Ajax_(programming) 19

[48] http://www.rubydoc.info/github/jnicklas/capybara/Capybara/Node 20

[49] http://www.rubydoc.info/github/jnicklas/capybara/Capybara/Node 21

[50] https://www.relishapp.com/cucumber/cucumber/docs/transforms 22

[51] https://github.com/rspec/rspec-core 22

[52] https://github.com/rspec/rspec-expectations 22

[53] https://github.com/rspec/rspec-mocks 22

[54] https://github.com/rspec/rspec-rails 22

[55] http://guides.rubyonrails.org/testing.html 22

[56] https://github.com/metaskills/minitest-spec-rails 22

[57] http://sass-lang.com/ 25

[58] http://en.wikipedia.org/wiki/XPath 29

[59] https://github.com/cucumber/cucumber/wiki/Hooks 31

[60] https://github.com/codahale/bcrypt-ruby 33

[61] http://jqueryui.com/

37