1 RUBY ON RAILS

1.1 INTRODUZIONE A ROR

1. Perchè Rails, rimando ad articolo

1.1.1 *Ruby*

Ruby è un linguaggio di programmazione open-source, general-purpose e orientato agli oggetti ideato da Yukihiro Matsumoto. La prima pubblicazione del linguaggio risale al 1995 ed attualmente è alla versione 2.1. Ruby è fortemente orientato alla produttività, permette di scrivere codice essenziale, con un netto risparmio di linee di codice rispetto a linguaggi tipati staticamente come Java o C# e offre una sintassi ricca e versatile.

La comunità di Ruby è estremamente attiva e dinamica e contribuisce in maniera attiva all'individuazione di bug e al miglioramento continuo del linguaggio. Non è l'obbietto di questa tesi addentrarsi nelle particolarità sintattiche e semantiche di Ruby, ma ritengo sia utile fornire qualche riferimento per l'apprendimento del linguaggio.

«Se intraprendete lo studio di un kōan e vi ci dedicate senza interrompervi, scompariranno i vostri pensieri e svaniranno i bisogni dell'io. » (Hakuin, Orategama)

I Ruby Koans[1], "koan" è la pronuncia giapponese dei caratteri cinesi, sono una collezione di esercizi su Ruby e permettono lo studio del linguaggio comprendendo la maggior parte strutture sintattiche, degli elementi semantici e delle strutture dato. L'utente è guidato tramite nell'apprendimento in puro stile TDD, in maniera semplice, leggera ed irriverente; terminato il corso si acquisisce una buona consapevolezza del linguaggio e confidenza con lo stile di Ruby.

Sempre nello stile tracciato da i koan, tenuti in massima considerazione all'interno della comunità di utenti di Ruby, ho approfondito la conoscenza del linguaggio e di RoR tramite il portale Ruby Monk[2] che fornisce numerosi tutorial interattivi ed esercizi riassuntivi per ogni argomento; ogni lezione è completabile attraverso il proprio browser internet e fornisce numerosi consigli e indicazioni allo studente.

1 I Koans sono disponibili su http://rubykoans.com/.
2 https://rubymonk.com/

1.1.2 *Don't Repeat Yourself*

Il principio di mantenere il proprio senza ripetizioni e ben fattorizzato è un'ottima pratica di programmazione. Un progetto DRY permette agli sviluppatori di modificare il codice più semplicemente; una funzionalità descritta in un numero ridotto di unità di compilazioni 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, helpers 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 poter 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", che si occupa di visualizzare file con estensione 'erb', dei parametri in maniera opzionale.

Listato 1: Visualizzazione di un vista parziale.

```
<%= render
  :partial => 'posts/logged_user_post_actions',
  :locals => {
    :current_user => current_user,
    :post => post
  }
%>
```

Listato 2: Vista parziale.

```
<% if current_user %>
  <div class='post_actions'>
    <%= edit_post_image_link post %>
    <%= remove_post_image_link post %>
  </div>
<% end %>
```

Inoltre, data una vista parziale che visualizza un singolo elemento, è anche possibile passare al metodo render una collezione di oggetti sulla quale iterare, fornendo opzionalmente anche un separatore.[3]

La piattaforma RoR fornisce anche numerosi Helper. Un Helper è descritto da un insieme di funzionalità, definite all'interno di un modulo, 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 sono solo alcuni dei problemi risolvibili tramite gli helper definiti nell'API di RoR.

3 I metodi e le librerie in RoR tipicamente si prestano ad una grande varietà di usi e comportamenti, per una descrizione più precisa è possibile consultare la documentazione online. http://api.rubyonrails.org/classes/ActionView/PartialRenderer.html

Listato 3: Il metodo 'truncate', appartenente al modulo TextHelper.

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

Come già osservato durante l'uso delle viste parziali, le librerie offerte sono ricche di parametri opzionali, permettono l'uso di blocchi definiti dall'utente e sono ben documentate: ogni metodo presente è ampiamente descritto e offre diversi esempi per semplificarne l'uso.

Listato 4: Un esempio dell'uso delle funzionalità di FormHelper.

```
<%= form_for(@post) do |f| %>
  ...
  <div id="title_field">
  <%= f.label :title, 'Titolo' %>
  ...
  <%= f.text_field :title, :size => 50 %>
  </div>
  ...
<% 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.

1.1.3 *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[4], un formato di serializzazione facilmente leggibile e scrivibile, e Ruby stesso. Lo sviluppatore inoltre può anche sfruttare i numerosi tool a linea di comando per la prima generazione delle risorse, utili anche per avere una traccia contenente le impostazioni più plausibili per una nuova applicazione.

Listato 5: Frammento in YAML relativo alla configurazione del modello.

```
default: &default
adapter: postgresql
encoding: unicode
host: localhost
pool: 5
```

4 Per informazioni si rimanda al sito ufficiale.http://www.yaml.org/

```
6
7  development:
8  <<: *default
9  database: rblog_development
10 username: xblog
11 password: ...
```

1.1.4 *L'interpretazione di RoR del pattern MVC*

Il modello

L'interpretazione di RoR del modello dell'architettura MCV prende spunto dal patter Active Record introdotto da Martin Fowler. I record attivi introducono un livello di astrazione fra gli strumenti di persistenza ed i controlli che gestiscono e manipolano il dominio; i dati sono astratti a 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 che le funzionalità che ne descrivono il comportamento attraverso i metodi d'istanza.

Il database, ed in generale qualunque sistema garantisca la persistenza della nostra applicazione, non è acceduto direttamente ma tramite il processo ORM. 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 delineare in maniera netta la separazione fra modello e controlli.

Oltre a rappresentare i dati, è possibile sfruttare Ruby per individuare relazioni di ereditarietà fra i tipi esistenti, validare i dati attraverso l'invocazione di metodi e definire le proprie interrogazioni attraverso un apposita libreria.

Listato 6: Esecuzione di una semplice query in Ruby.

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

I controlli

Essendo i controlli gli elementi di interconnessione fra il modello e le viste è difficile per lo sviluppatore mantenerne l'implementazione semplice. Proprio per semplificare la definizione ed il mantenimento sono presenti alcuni accorgimenti in RoR.

La configurazione delle richieste instradabili è specificata in file separato "*routes.rb*", all'interno del quale è possibile configurare le richieste HTTP soddisfabili e definire una gerarchia delle entità manipolate dall'applicazione secondo i principi di un'architettura REST.

Listato 7: 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]
```

Un altro accorgimento apprezzato durante lo sviluppo è la possibilità di definire dei filtri per i controlli. Per ogni controllo sono previsti alcuni stati a cui è possibili attribuire eventi, i filtri per l'appunto. Sfruttando questa semplice tecnica è possibile fattorizzare alcuni comportamenti comuni all'interno dei controlli di una stessa entità, come ad esempio la verifica delle autorizzazione per alcune operazioni sensibili.

Listato 8: L'uso dei filtri in RBlog.

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

Le viste

Le viste in RoR sono definibili introducendo nei file HTML elementi dinamici contenenti espressioni scritte in Ruby. ERB, acronimo per Embedded Ruby, permette l'inserimento di codice da processare all'interno del server web prima di fornire la risposta al client.

La definizione di viste dinamiche risulta immediata, conoscendo sia Ruby che HTML tramite solo tre tipi di delimitatori, che variano esclusivamente nel tipo di output generato.[5] Le viste scritte tramite ERB sono facilmente leggibili grazie alla caratteristiche di Ruby, alla sintassi minima e alle numerose funzionalità presente negli Helper e librerie del linguaggio.

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.

Listato 9: Frammento di vista relativo alla visualizzazione di un singolo post.

```
<div class='post'>
  <p class='post_title'>
    <%= link_to @post.title, @post %>
  </p>
  <p class='post_detail'>
    <%= author_detail(@post) %>
    </br>
    <% post_details(@post).each do |detail| %>
      <%= detail %>
    <% end %>
  </p>
    <p class='post_content'>
    <%= @post.body %>
  </p>
```

5 Per maggiori informazioni relative alle viste e alla definizione di pagine dinamiche tramite ERB si rimanda alla seguente guida: http://guides.rubyonrails.org/layouts_and_rendering.html

```
15    <%= render
16      :partial => 'logged_user_post_actions',
17      :locals => {:current_user => current_user, :post => @post}
18    %>
19  </div>
```

Il testing

La propensione alla verifica dell'applicazione non è un elemento proprio dell'architettura MVC, ma influisce comunque notevolmente sul processo di sviluppo.

Tramite RoR è possibile gestire l'intero stack di un'applicazione web, test inclusi.[6] Sfruttando RSpec, strumento eletto a standard defacto, è possibile testare il modello attraverso i record attivi ed anche verificare i propri controller, definendo i parametri HTTP e dati.

Da Ruby 1.9 è incluso la libreria MiniTest, 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.[7][8] Esistono numerose gemme sviluppate da terze parti per aggiungere funzionalità e semplificare il processo di testing, ad esempio per la pulizia dei database utilizzati DatabaseCleaner è molto diffusa, al pari di Factory Girl[9] per la popolazione del modello.[10]

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.
Rail fornisce inoltre il comando Rake per poter gestire al meglio l'esecuzione selettiva delle proprie librerie di test.[11]

1.1.5 *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 e del suo "*ecosistema*" che hanno contribuito che lo sviluppo si svolgesse in maniera lineare permettendo di scrivere codice in maniera produttiva e concentrando la propria attenzioni all'indagine sui test di accettazione piuttosto che a problematiche di contorno.

6 http://guides.rubyonrails.org/testing.html
7 https://github.com/seattlerb/minitest
8 Paragonabili alla annotazioni *@After*, *@Before*, *@AfterClass* e *@BeforeClass* in JUnit 4.x.
9 https://github.com/DatabaseCleaner/database_cleaner
10 https://github.com/thoughtbot/factory_girl
11 http://guides.rubyonrails.org/testing.html#rake-tasks-for-running-your-tests

A differenza di altre piattaforme, in cui sono presenti strumenti esterni per supportare la compilazione e la risoluzione delle dipendenze come Maven e Gradle, Ruby sfrutta un proprio meccanismo, definendo delle gemme. Ogni gemma rappresenta una libreria, ha un nome, versione ed architettura di riferimento.

Ogni applicazione è caratterizzata da una risorsa chiamata Gemfile, un semplice script in Ruby, che rappresenta l'insieme delle dipendenze e delle gemme da includere nel proprio progetto.[12] 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 10: Frammento del Gemfile di RBlog.

```
1 source 'https://rubygems.org'
2
3 group :development, :test do
4   gem 'cucumber-rails', :require => false    gem 'rspec-rails'
5   gem 'capybara'
6   gem 'poltergeist'
7   gem 'database_cleaner'
8 end
```

Fra le gemme utilizzate, la più preziosa è Spring. La libreria permette di caricare in anticipo le modifiche fatte al codice sorgente dell'applicazione in maniera che sia sempre in esecuzione la versione più recente, evitando all'utente di dover riavviare l'esecuzione manualmente ad ogni cambiamento. :)

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

In RoR è previsto il meccanismo delle migrazioni che fornisce uno strumento per mantenere lo schema del modello consistente durante il processo di sviluppo in maniera semplice e lineare. Ogni variazione alla struttura del modello è tradotto in una migrazione, che definisce l'insieme delle operazioni compiute a basso livello, l'aggiunta di colonne, la rimozione di un vincolo etc., ed attribuisce una versione all'operazione; le stesse informazioni sulle migrazioni sono mantenute nel modello e tengono traccia dei cambiamenti apportati e dello stato attuale dello schema, permettendo di mantenere tutte le componenti dell'architettura MVC consistenti fra loro.

1.2 RUBYMINE

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

12 https://rubygems.org/ è il servizio di hosting di riferimento.

JetBrains, sviluppatori anche di IntelliJ IDEA, Android Studio, ReSharper per citare i più conosciuti, supporta le feature di Rails 4.1 e Ruby 2.1, ovvero le versioni più recenti.

L'esperienza con RubyMine è stata ottima: durante lo sviluppo sono stati rilevati raramente problemi ed integra perfettamente tutte le funzionalità a riga di comando offerte da Rails; Ruby, HTML, JavaScript, JQuery, CoffeeScript, SCSS sono solo alcuni dei linguaggi supportati nell'IDE e per tutti sono definite varie strategie di refactoring ed è possibile analizzarne le relative risorse tramite un potente tool di analisi del codice.

Nell'IDE sono presenti diversi plugin per l'integrazione con strumenti di terzi, come ad esempio Cucumber, Git[13] e SSH. RubyMine è un prodotto curato nei dettagli, professionale ed allo stesso tempo adatto anche agli utenti alle prime armi; ma soprattutto permette l'implementazione di un'applicazione in Rails praticamente senza abbandonare l'ambiente di sviluppo.

1.3 HELLO RBLOG!

L'obbiettivo di questa prima funzionalità 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.

1.3.1 *Cucumber*

Il primo tassello necessario per l'implementazione dei test di accettazione su un'applicazione web sviluppata in RoR è Cucumber.

Cucumber è un framework per il supporto al BDD, Behaviour Driven Development, che permette di scrivere i propri test in maniera leggibile, elegante e mantenibile. Il framework è disponibile anche in Java, .Net, Python, PHP e molti altri linguaggi e piattaforme.

Le funzionalità

Listato 11: 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.

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

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 gli 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 secondo il formato "*Per* <beneficio>, *come* <ruolo>, *vorrei* <obbiettivo, desiderio>". 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 listato 11, 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 ereditariet. L'etichetta è stata introdotta per poter eseguire in maniera selettiva gli scenari della prima funzionalità.

Listato 12: Comando per l'esecuzione degli scenari relativi alla feature "Hello RBlog!"

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

E' possibile introdurre un numero arbitrario di etichette per ciascun elemento, sia 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 13: 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 11 ed introduce la sintassi di Cucumber. Gli scenario sono caratterizzati da un titolo, che riassume il comportamento e da un insieme di passi. Seguendo la logica del BDD, ogni passo può alternativamente 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 dall'utente 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 lo stato 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 14: 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 14 è effettuato sia sulla pagina dell'autore che sulla pagina dell'abstract.

Listato 15: Testo generato dallo scenario 14

```
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.[14] Gli autori, gli stessi di Cucumber, descrivono il linguaggio come Business Readable DSL, definizione introdotta da Martin Fowler nel 2008.[15]

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 rispetto ad uno anche scrivibile, infatti il principale obbiettivo di Business Readable DSL è stabilire un canale di comunicazione arricchente fra le diverse parti che partecipano allo sviluppo.

I18n

Cucumber supporta attualmente 40 lingue, numero in rapida crescita secondo gli sviluppatori.[16]

In Cucumber 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.

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, è possibile navigare dalla definizione all'implementazione del passo ed è fornita un'interfaccia grafica per l'esecuzione dei test, configurabile in funzione di tag, file delle funzionalità da includere ed altri parametri.

1.3.2 *Capybara*

Capybara 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.

14 Maggiori informazioni relative alla sintassi sono reperibili alla pagina https://github.com/cucumber/cucumber/wiki/Gherkin

15 L'articolo completo di Fowler è consultabile sul suo blog http://martinfowler.com/bliki/BusinessReadableDSL.html

16 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.

Capybara nasconde all'utente i dettagli tecnici della navigazione tramite primitive di funzioni semplici, intuitive e versatili. A differenza di Selenium, le cui primitive si interfacciano con aspetti a basso livello, le funzionalità di Capybara sono astratte e mantengono la stessa prospettiva di un test effettuato manualmente da un utente.

La configurazione di Capybara è effettuata tramite un breve script in Ruby dove è necessario impostare le proprie preferenze. Innanzitutto è necessario scegliere quale browser web sarà utilizzato ed il relativo driver per Capybara importando la libreria nel file "*features/support/env.rb*".

Listato 16: Dipendenze all'interno dello script di configurazione.

```
1 require 'cucumber/rails'
2 require 'capybara'
3 require 'capybara/cucumber'
4 require 'capybara/rspec'
5 require 'capybara/poltergeist'
```

Infine, se necessario, è possibile configurare il driver scelto: la struttura di Capybara permette l'uso di diversi driver e browser, senza modificare le primitive offerte. Tutti i driver 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 Poltergeist.

Listato 17: Configurazione di Poltergeist

```
1  Capybara.default_driver = :poltergeist
2  Capybara.register_driver :poltergeist do |app|
3    options = {
4      :js_errors => true,
5      :timeout => 120,
6      :debug => true,
7      :phantomjs_options => ['--load-images=yes', '--disk-cache=
           false'],
8      :inspector => true,
9    }
10   Capybara::Poltergeist::Driver.new(app, options)
11 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;
- :debug reindirizza l'output dell'esecuzione in modalità debug verso STDERR.

1.3.3 *PhantomJS*

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

E' particolarmente adatto all'esecuzione automatica di applicazioni web in quanto non è necessario che il sistema sia dotato di un framework per la gestione di GUI, funzionalità spesso mancante sui server che effettuano l'integrazione continua.

Con PhantomJS è possibile far fallire i propri test in funzione di errori su codice JavaScript, funzionalità non presente su altri browser, grazie all'opzione :js_errors precedentemente descritta.[17]

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 perché da ottimi risultati nella gestione di strumenti asincroni come AJAX, è estremamente veloce e molto accurata nella gestione di falsi positivi, come ad esempio il click su eventi esistenti nel DOM di una pagina HTML ma non visibili.[18][19]

1.3.4 *Implementazione dei passi*

La struttura dei passi in Capybara 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.

Listato 18: Implementazione del passo "apro RBlog".

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

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

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

19 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.

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, Capybara verifica che sia presente un'implementazione per ogni passo definito tramite Cucumber ed applica le istruzioni associate. E' necessario che per ogni passo in linguaggio naturale esista un'unica espressione regolare corrispondente.

Listato 19: 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.[20]

La struttura di Capybara

L'implementazione di passo è definibile all'interno di un blocco[21] e non esiste alcuna limitazione né controllo riguardo alla tipologia di espressioni che vengono eseguite durante il passo.

In Capybara esistono numerose funzionalità per la gestione degli elementi, indicati come nodi. La gerarchia dei nodi 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 Element e Document: gli oggetti delle classi figlie condividono gli stessi metodi sfruttando i metodi presenti nei moduli Finders, Matchers e Actions. A differenza dei nodi semplici è quindi possibile compiere azioni su di essi;
- la classe Element rappresenta un singolo elemento all'interno del DOM della pagina;
- la classe Document rappresenta i documenti HTML nella loro interezza.

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

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

21 In Ruby è possibile attribuire ad ogni metodo un blocco contenente del codice da eseguire durante l'esecuzione. I blocchi sono paragonabili ad espressioni lambda, ma non sono elementi di prim'ordine del linguaggio.

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 ed 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.[22]

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.

Listato 20: Istruzioni equivalenti per l'accesso all'attributo title di una pagina HTML.

```
1 page.title
2 page[:title]
3 page.has_title? "xyz"
4 page.has_no_title? "zyx"
```

Navigare all'interno del sito

Nel passo 18 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.[23]

Listato 21: 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
```

Il metodo visit, utilizzato per aprire la pagina iniziale del blog, non verifica la presenza all'interno della pagina di un collegamento verso

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

23 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.

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".[24] Il metodo find_link 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 lancia un eccezione nel caso in cui la ricerca non dovesse individuare esattamente un'elemento. Per completare l'evento, semplicemente si invoca il metodo "click" sul nodo restituito.

A differenza del passo 18, dove il comportamento non varia in funzione di parametri, nel passo 21 viene selezionato il collegamento che coincide con il valore di "page_name", tramite un blocco con un singolo parametro. Il plugin di Cucumber per RubyMine 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 trasformazioni per convertire il valore ad un nuovo tipo tramite il metodo Transform.[25]

Definizione delle asserzioni con RSpec

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

- RSpec-Core fornisce la struttura per la definizione di funzionalità e scenari per il BDD;
- RSpec-Expectations è una libreria di metodi per definire asserzioni;
- RSpec-Mocks è un framework per l'implementazione di stub, oggetti mock, verifiche sull'invocazione dei metodi e dell'interazione fra oggetti;
- RSpec-Rails è un framework per la definizione di test sulle componenti che definiscono un'applicazione RoR, come il modello,

24 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.

25 Per maggiori dettagli consultare la documentazione di Cucumber https://www.relishapp.com/cucumber/cucumber/docs/transforms.

i controlli e le viste ma anche gli aiutanti e l'instradamento delle richieste.

All'interno della libreria di Capybara non sono presenti le funzionalità per la verifica di asserzioni, è quindi necessario sfruttare librerie terze. Oltre a RSpec è possibile l'integrazione all'interno di unit-test, Test::Unit[26] in Rails, oppure con le asserzioni definite in MiniTest::Spec[27].

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.[28]

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 22: Sintassi 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, invocato durante l'esecuzione del metodo, che viene verificato dai matcher indicati.

Un matcher in RSpec è un metodo e fornisce un risultato booleano in funzione dell'operazione implementata. L'uso corretto dei matcher è come argomenti dei metodi "to" e "not_to".

Listato 23: 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 14 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 24: Implementazione di un'asserzione parametrica.

26 Maggiori informazioni per la definizione di test in Rails sono disponibili alla pagina http://guides.rubyonrails.org/testing.html.

27 Maggiori informazioni sono contenute alla pagina del progetto https://github.com/metaskills/minitest-spec-rails.

28 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.

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

Nello scenario 14 è richiesto il confronto di un parametro, definito attraverso l'espressione regolare, ed il titolo della pagina.[29] 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 25: Implementazione del passo "apro RBlog".

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

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

1.4 INTRODUZIONE DEL CSS

Obbiettivo
Sfruttare ATDD per verificare l'Accessibilità
Principi di accessibilità

1.4.1 *SCSS*

1.5 DEFINIZIONE DEL MODELLO

- Tag per la pulizia piuttosto che sfruttare un una gemma come FactoryGirl o DatabaseCleaner - http://www.railsonmaui.com/tips/rails/capybara phantomjs-poltergeist-rspec-rails-tips.html - http://robots.thoughtbot.com/how-we-test-rails-applications

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 per il product owner, che tramite gli scenari può seguire lo sviluppo del prodotto.

Gli scenari devono essere indipendenti fra loro, l'ordine di esecuzione non deve inficiare il risultato. Mantenere delle dipendenze funzionali all'interno nella libreria di testing può introdurre diverse

29 Capybara::DSL::page è un getter e restituisce la sessione correntemente sotto test. Una sessione in Capybara rappresenta l'interazione dell'utente con il sistema, maggiori dettagli sono presenti nella relativa sezione.

problematiche all'aumentare della cardinalità e della complessità o alla variazione dello condizioni d'esecuzione, introducendo ad esempio l'esecuzione in parallelo di più scenari.

Per semplificare la lettura degli scenari ed il riuso dei passi è conveniente sfruttare il prefisso "And".

1.6 definizione delle viste

find, find_link, find_content, xpath, contesto di ricerca
metodi sulla pagina

1.7 introduzione della funzionalità di ricerca

http://www.rubydoc.info/github/jnicklas/capybara/Capybara/Node/Finders#find-instance_method

Asincronia in Capybara https://blog.codecentric.de/en/2013/08/cucumber-capybara-poltergeist/

1.8 introduzione dell'autenticazione

1.9 le sessioni in capybara

1.10 mantenibilità degli acceptance tests

http://www.elabs.se/blog/15-you-re-cuking-it-wrong

http://eggsonbread.com/2010/09/06/my-cucumber-best-practices-and-tips/

https://blog.engineyard.com/2009/15-expert-tips-for-using-cucumber

http://blog.codeship.io/2013/05/21/testing-tuesday-6-top-5-cucumber-best-practices.html

https://github.com/strongqa/howitzer/wiki/Cucumber-Best-Practices

1.11 conclusione