

UNIVERSITÀ DEGLI STUDI  
DI GENOVA

Facoltà di Scienze Matematiche Fisiche Naturali

Corso di Laurea Magistrale in Informatica

**Test di accettazione in  
framework MVC a confronto**

Relatore:  
Prof. Maura Cerioli  
Correlatore:  
Prof. Davide Ancona

Tesi magistrale di:  
Mattia Barrasso  
Matricola N° 3301135

Anno Accademico 2013/2014



# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	L'architettura Model View Controller . . . . .	2
1.2	I test di accettazione . . . . .	3
1.3	Il caso di studio . . . . .	5
1.4	Sommario . . . . .	7
<b>2</b>	<b>Ruby On Rails</b>	<b>9</b>
2.1	Ruby On Rails ~ RoR . . . . .	9
2.1.1	Ruby . . . . .	9
2.1.2	RubyMine . . . . .	14
2.2	L'interpretazione di RoR del pattern MVC . . . . .	15
2.2.1	Il modello . . . . .	15
2.2.2	I controlli . . . . .	15
2.2.3	Le viste . . . . .	17
2.2.4	Il testing . . . . .	18
2.2.5	Peculiarità . . . . .	19
2.3	Hello RBlog! . . . . .	23
2.3.1	Cucumber . . . . .	23
2.3.2	Capybara . . . . .	25
2.3.3	Implementazione dei passi . . . . .	27
2.4	Introduzione del CSS . . . . .	32
2.4.1	"CSS with superpowers" . . . . .	32
2.4.2	Testare il css . . . . .	33
2.4.3	Il contesto degli scenari . . . . .	35
2.4.4	Debug con Capybara . . . . .	36
2.4.5	XPath . . . . .	36
2.5	Definizione del modello . . . . .	37
2.5.1	Dipendenze . . . . .	37
2.5.2	Gestione dei form . . . . .	39
2.6	Login & Autorizzazione . . . . .	40
2.6.1	Black-box Testing . . . . .	40
2.6.2	Manutenibilità . . . . .	41
2.7	Asincronia . . . . .	43
2.7.1	JavaScript . . . . .	43

2.7.2	JQueryUI	44
2.7.3	Scenari sull'auto-completamento	45
<b>3</b>	<b>Spring</b>	<b>49</b>
3.1	Spring MVC	49
3.1.1	Spring Tool Suite	49
3.2	L'interpretazione di Spring del pattern MVC	50
3.2.1	Il modello	50
3.2.2	I controlli	55
3.2.3	Le viste	58
3.2.4	Il testing	61
3.2.5	Peculiarità	61
3.3	Hello SBlog!	63
3.3.1	Cucumber-JVM	63
3.3.2	Selenium	64
3.3.3	Implementazione dei passi	65
3.4	Introduzione del CSS	68
3.4.1	Testare il css	68
3.4.2	Debug con Selenium	71
3.5	Definizione del modello	72
3.5.1	Dipendenze	72
3.5.2	Gestione dei form	73
3.6	Login & Autorizzazione	75
3.7	Asincronia	77
3.7.1	JavaScript	77
3.7.2	Scenari sull'auto-completamento con JQuery UI	78
<b>4</b>	<b>MVC 5</b>	<b>79</b>
4.1	ASP.NET MVC 5	79
4.1.1	Visual Studio 2013	79
4.2	L'interpretazione di ASP.NET del pattern MVC	80
4.2.1	Il modello	80
4.2.2	I controlli	82
4.2.3	Le viste	83
4.2.4	Peculiarità	85
4.3	Hello CSBlog!	86
4.3.1	SpecFlow	86
4.3.2	Coypu	87
4.3.3	Implementazione dei passi	88
4.4	Introduzione del CSS	92
4.5	Definizione del modello	93
4.5.1	Dipendenze	93
4.5.2	Gestione dei form	94
4.5.3	Debug con Specflow	101
4.6	Login & Autorizzazione	102
4.6.1	Accesso ai singoli attributi	102

4.7	Asincronia . . . . .	103
4.7.1	JavaScript . . . . .	103
4.7.2	Scenari sull'auto-completamento con JQuery UI . . . . .	104
<b>5</b>	<b>Conclusione</b>	<b>106</b>
5.1	I framework . . . . .	106
5.1.1	Produttività . . . . .	106
5.1.2	Il modello . . . . .	110
5.1.3	I controlli . . . . .	115
5.1.4	Le viste . . . . .	117
5.2	ATDD . . . . .	120
5.2.1	BDD . . . . .	120
5.2.2	Web Automation . . . . .	121

# Capitolo 1

## Introduzione

Nell'articolo "A comparative Study of Maintainability of Web Application on J2EE, .NET and Ruby on Rails"~[1] scritto da Look Fang Fang Stella, Stan Jarzabek and Bimlesh Wadhwa nel 2008 viene discusso il processo di introduzione di una nuova funzionalità sul codice esistente di un'applicazione Web e delle ripercussioni di questa operazione nelle diverse piattaforme. Ruby on Rails, Spring e .NET condividono il pattern architetturale Model View Controller, descritto in seguito, e si prestano dunque ad un confronto alla pari: dall'articolo emerge come il neonato Ruby on Rails <sup>1</sup>, si presentasse fin da subito come una piattaforma moderna ed adeguata alla metodologia Agile, offrendo supporto ai relativi processi di sviluppo. Grazie ad una struttura più lineare, alla semplicità di definizione dei test e combinando il potere espressivo di Ruby, gli autori hanno individuato in RoR il framework che ha il minor impatto sul progetto per numero di linee di codice modificate ed al contempo mantiene il codice maggiormente comprensibile e di conseguenza più semplicemente modificabile.

Stella, Jarzabek e Wadhwa concludono il proprio lavoro segnalando come il campo delle applicazioni web sia in continuo fermento ed in rapida evoluzione ed assumono anche che nuovi framework potrebbero emergere in futuro ed essere meritevoli di attenzione.

L'obiettivo di questa tesi è di riprendere la prospettiva utilizzata per lo sviluppo di applicazioni web dai tre ricercatori dell'università di Singapore, analizzando tre framework, scelti in funzione di diffusione, novità e particolarità ma concentrando l'attenzione sull'implementazione di test di accettazione automatici piuttosto che alla modificabilità. I framework scelti sono Ruby on Rails, Spring MVC e ASP.NET MVC5. Nel corso della tesi verrà trattata l'interpretazione data da ciascun framework del pattern Model View Controller e documentato lo sviluppo delle applicazioni web in funzione dell'implementazione dei test di accettazione automatici.

---

<sup>1</sup>L'articolo è stato pubblicato pochi anni dopo il rilascio della prima versione.

Al fine di focalizzare l'attenzione sullo sviluppo e sulle problematiche dei test di accettazione è stato scelto di sviluppare una semplice applicazione web, riprendendo l'esempio del blog proposto nell'articolo in forma semplificata. Sono state sviluppati tre progetti, che offrono le stesse funzionalità e condividono l'aspetto e le funzionalità dell'interfaccia grafica, presentando però delle ovvie differenze relative all'implementazione. Inoltre è stato scelto di sfruttare il più possibile nuove tecnologie per verificare che gli strumenti per la definizione dei test di accettazione siano al passo con un settore, quale è lo sviluppo di applicazioni web, molto dinamico e variegato. Nel corso della tesi verranno documentati gli strumenti scelti, verificandone le potenzialità nei diversi scenari di test, effettuando confronti dove possibile ed evidenziando pregi o carenze rispetto alle controparti.

## 1.1 L'architettura Model View Controller

MVC è l'acronimo per Model-View-Controller, un pattern architetturale per l'implementazione di interfacce utente, e nel caso specifico di questa tesi, di applicazioni Web.

Il principale concetto che definisce l'architettura MVC è la separazione sia concettuale che pratica delle componenti che definiscono un'applicazione web.

Il modello è la prima componente individuabile, descrive il dominio in maniera indipendente dalla logica e dalle interfacce offerte all'utente; in maniera più specifica, nel contesto delle applicazioni web, il modello è definito da uno o più strumenti per garantire la persistenza, potenzialmente eterogenei.

Tramite le viste il sistema fornisce all'utente una rappresentazione delle informazioni presenti, e nel contesto di un'applicazione web rappresenta la pagina visualizzata tramite il browser, caratterizzata da elementi attivi che permettono l'interazione all'utente.

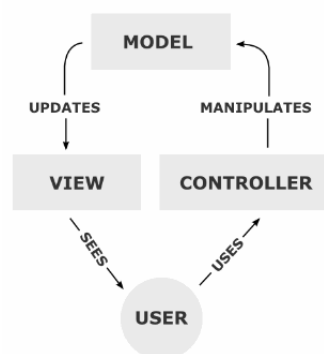


Figura 1.1: Dipendenze essenziali nel modello MVC.

I controlli rappresentano le entità che si occupano di ricevere ed interpretare le azioni compiute dall'utente. Nella definizione teorica del pattern MVC, come schematizzato nella figura 1.1, i controlli manipolano il modello ma non interagiscono con le viste, che si limitano a osservare lo stato del dominio e a reagire ad eventuali cambiamenti.



Figura 1.2: Model View Presenter

Al contrario, la maggior parte dei framework “MVC” attuali, danno un'interpretazione differente del ruolo del controllo, responsabile di interagire sia con il modello che con le viste, come avviene per il Model View Presenter, schematizzato in figura 1.2.

Il presenter rappresenta la componente fondamentale per l'interconnessione dell'architettura, è il gestore delle richieste effettuate dall'utente tramite le viste e interagisce con il modello per fornire una rappresentazione dei dati.

L'approccio classico del pattern MVC non è adatto alle applicazioni web moderne, in quanto le viste non possono reagire ai cambiamenti sul modello secondo il pattern dell'observer.

Nonostante il pattern Model View Presenter sia più rispondente alla realtà dello sviluppo web, in letteratura si fa sempre riferimento al pattern MVC e per questo nel seguito della tesi si discuterà tale pattern.

## 1.2 I test di accettazione

Il Behaviour Driver Development, BDD, è una metodologia di sviluppo che prevede la descrizione in linguaggio naturale delle funzionalità attese, utilizzando una terminologia che colga il comportamento del sistema, piuttosto che evidenziare dettagli tecnici dell'implementazione. Il fulcro del BDD è la definizione dei test di accettazione, usati per la descrizione “per esempi” delle funzionalità, appunto.

Per ciascun test di accettazione, è possibile descrivere più criteri di accettazione, o scenari, che descrivono in linguaggio naturale un caso specifico della funzionalità, ovvero un suo esempio d'uso specifico.

**Business Readable DSL** Il linguaggio utilizzato per descrivere le funzionalità e gli scenari nella tesi è Gherkin~[41], sviluppato per il framework Cucumber, che verrà presentato nel prossimo capitolo. Gli autori descrivono il linguaggio come Business Readable DSL~[40], definizione introdotta da Martin Fowler nel 2008.



L'obiettivo 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.

Il formato standard dei test in Gherkin è dato concatenazione di tre blocchi di passi:

- blocco "Given": una sequenza di condizioni (passi) che esprimono le condizioni iniziali o di setup del test;
- blocco "When": la definizione di uso della funzionalità descritta;
- blocco "Then": le condizioni osservabili, risultato dell'operazione;

Se le funzionalità dovessero essere leggibili solo da tecnici, potrebbe essere sufficiente la definizione dei passi in lingua inglese. Invece, per permettere la leggibilità a più utenti possibili, Gherkin supporta attualmente 40 lingue, numero in rapida crescita secondo gli sviluppatori.<sup>2</sup> E' così possibile definire i passi degli scenari con le parole chiave nella lingua scelta: ad esempio per la definizione delle funzionalità dei blog è stata utilizzata la lingua italiana<sup>3</sup>.

## ATDD

Acceptance Test Driven Development, ATDD, è una tecnica per produrre codice rilevante e utile ai fini dell'implementazione di una certa funzionalità. L'ATDD porta a livello di intero sistema i principi del TDD, Test Driven Development.

Infatti, lo sviluppo del sistema inizia con lo sviluppo dei test di accettazione, il cui sviluppo condiviso fra tutti gli stakeholder permette di catturare le specifiche del sistema. Nelle fasi successive dello sviluppo, i test potranno naturalmente essere impiegati anche per verificare la correttezza dell'implementazione e come test di regressione.

Secondo la Agile Alliance, gli acceptance test dovrebbero essere automatizzati per poter essere eseguiti spesso, come suggerito dalla pratica dell'integrazione continua.

A differenza della verifica dei test effettuata manualmente dagli utenti, i test automatici hanno il vantaggio di restituire un risultato in maniera deterministica per un certo requisito e permettere al team di concentrare meglio i propri sforzi.

Utilizzando la tecnica dell'ATDD sono prodotti test di accettazione che rappresentano anche la documentazione della funzionalità ed essendo eseguibili e verificabili, rappresentano la documentazione aggiornata del prodotto.

<sup>2</sup>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>.

<sup>3</sup>Per utilizzare uno dei pacchetti linguistici esistenti è necessario un header `"# language: xx"` all'interno dei file scritti in Gherkin.

### 1.3 Il caso di studio

Lo scopo della tesi è ripetere, a distanza di alcuni anni dall'articolo "A comparative Study of Maintainability of Web Application on J2EE, .NET and Ruby on Rails" il confronto fra le potenzialità attuali dei framework MVC Spring, Ruby on Rails e ASP.NET MVC. A tale scopo, come nell'articolo originale, si userà come caso di studio blog sviluppato sulle tre piattaforme.

Grazie allo sviluppo della stessa applicazione web sui diversi framework, sarà possibile comprendere l'interpretazione del pattern architetturale MVC e analizzare per ciascuna componente pregi e difetti, in un confronto alla pari.

Un elemento che inevitabilmente influenza un confronto fra strumenti di sviluppo è il processo adottato per la realizzazione delle applicazioni. Seguendo il trend moderno si è deciso di adottare uno stile ATDD. Pertanto, oltre all'implementazione dei blog, nel seguito indicati con i nomi RBlog, SBlog e CSBlog -rispettivamente per le versioni in Spring, Ruby on Rails e ASP.NET MVC5- sono stati definite dei test di accettazione e l'analisi del confronto è concentrata su questo aspetto.

Durante la fase di design dei test di accettazione è stato scelto di far coincidere a ciascuna funzionalità lo sviluppo di una ulteriore componente del blog, che richiedesse l'introduzione di una singola tecnologia, per isolare al meglio i vari aspetti:

1. la funzionalità chiamata "Hello \*Blog!", include gli scenari relativi al primo utilizzo del blog e alla navigazione fra pagine statiche. Per verificare tali scenari è necessario solo che l'applicazione sia eseguibile e opportunamente configurata;
2. la funzionalità "Introducendo il CSS" richiede la definizione dei fogli di stile per il blog;

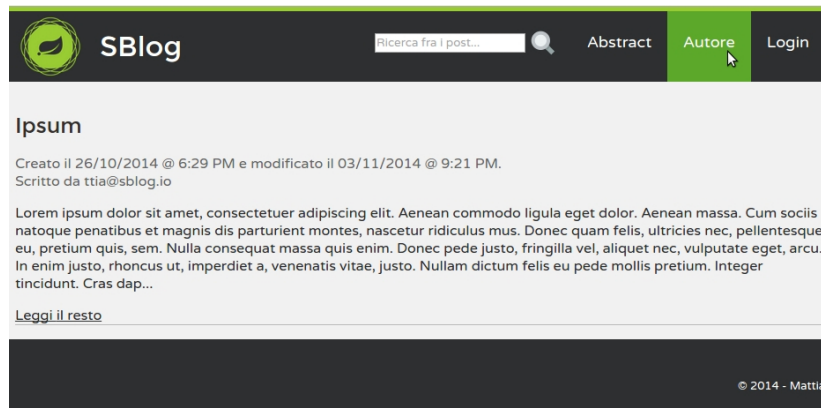



Figura 1.3: Schermata di SBlog, dopo la definizione dei fogli di stile.

3. “Gestione dei post” prevede che tramite l’interfaccia dell’applicazione web sia possibile gestire i post presenti sul blog, effettuando le operazioni di creazione, cancellazione e modifica di un articolo. Per soddisfare gli scenari dichiarati è necessario che il modello dell’applicazione sia configurato ed integrato con le altre componenti;



The screenshot shows the 'Scrivi un nuovo post' (Write a new post) form in the SBlog application. The form has a title field and a post content field. Two validation errors are displayed in red text: '2 errori hanno impedito il salvataggio.' (2 errors have prevented saving.) and 'Il titolo è già presente.' (The title is already present.) for the title field, and 'Il post deve essere almeno di 5 caratteri.' (The post must be at least 5 characters.) for the post content field. The form is set against a light gray background with a dark header bar containing the SBlog logo, a search bar, and navigation links for Abstract, Autore, and ttia@sblog.io- Esci.

Figura 1.4: Schermata di SBlog, implementazione della validazione dei post.

4. “Navigazione dei post” richiede che sia possibile leggere i post ed effettuare l’espansione dell’anteprima sfruttando i collegamenti dinamici introdotti;
5. “Easter Egging” introduce un comportamento faceto all’interno dell’applicazione sfruttando JavaScript per modificare il DOM della pagina;

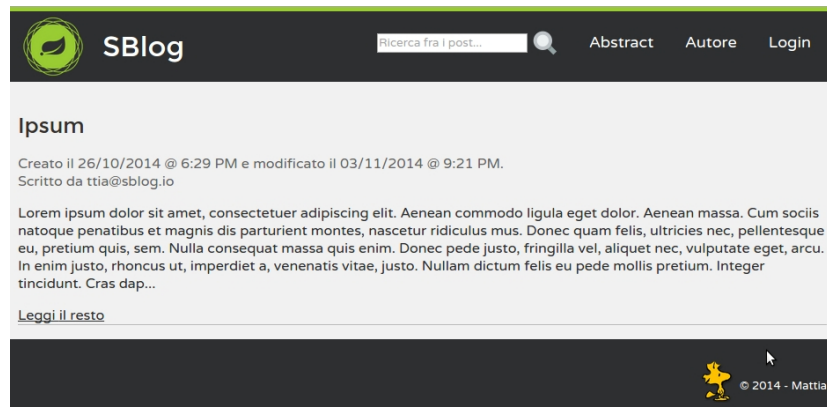


Figura 1.5: Schermata di SBlog, “easter egg”.

6. “Auto-completamento della ricerca” definisce la possibilità di ricercare dei post all’interno del sito, richiedendo l’uso di un widget asincrono per effettuare l’auto-completamento della ricerca;

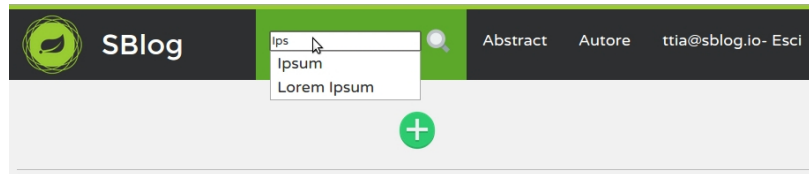


Figura 1.6: Schermata di SBlog, implementazione della ricerca con auto-completamento.

7. “Autenticazione su \*Blog” definisce le funzionalità di login e logout e i meccanismi per la verifica delle autorizzazioni.

Le funzionalità appena descritte e i relativi scenari saranno utilizzati per la definizione dei test di accettazione automatici per ogni progetto.

Per selezionare il miglior stack di strumenti da utilizzare per i test sono stati individuati alcuni parametri per effettuare la scelta:

- preferenza per strumenti implementati nello stesso linguaggio del framework, per minimizzare problemi di integrazione e favorire lo sviluppo tramite ATDD;
- utilizzo di strumenti gratuiti, preferibilmente open-source;
- utilizzo di strumenti supportati da una comunità di utenti e correntemente sotto sviluppo;

L’implementazione dei test di accettazione sarà utilizzata per ottenere informazioni sulle potenzialità degli strumenti utili per automatizzare la verifica di applicazioni web moderne. Inoltre, mantenendo per tutti i progetti le stesse funzionalità e scenari, sarà più semplice eseguire una comparazione fra gli strumenti utilizzati.

## 1.4 Sommario

Nei successivi capitoli è trattato lo sviluppo dei progetti e dell’implementazione dei test di accettazione automatici.

Nel secondo capitolo è descritto lo sviluppo di RBlog utilizzando Ruby on Rails e lo stack di strumenti per il testing, definito da Cucumber, Capybara e RSpec.

Nel terzo capitolo è trattato lo sviluppo di SBlog utilizzando Spring MVC per lo sviluppo dell’applicazione e Cucumber-JVM, Selenium e JUnit per lo sviluppo dei test.

L’ultimo progetto ad essere descritto è CSBlog nel quarto capitolo, sviluppato tramite il framework ASP.NET MVC5 e lo stack SpecFlow, Coypu e NUnit.

Infine, l'ultimo capitolo riassume quanto osservato attraverso l'esperienza di sviluppo dei diversi progetti, effettuando un confronto sia dell'interpretazione data per ciascuna componente del pattern MVC dai framework, sia delle potenzialità dei diversi strumenti utilizzati per l'implementazione dei progetti e dei test di accettazione.

## Capitolo 2

# Ruby On Rails

### 2.1 Ruby On Rails ~ RoR

In questo capitolo è trattato lo sviluppo di RBlog attraverso Ruby on Rails, RoR, un framework per lo sviluppo di applicazioni web scritto in Ruby la cui architettura è definita secondo il pattern MVC; l'implementazione del nostro caso di studio è stata compiuta sfruttando il metodo Acceptance Test-Driven Development, nel seguito ATDD.

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 2.2 è introdotto RoR e la sua architettura, descrivendone le particolarità. Le sezioni dalla 2.3 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.

#### 2.1.1 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~[2]. E' un linguaggio di programmazione orientato agli oggetti puro, tipato dinamicamente ed interpretato. A differenza di Java ed altri linguaggi ad oggetti, ogni tipo è definito da una classe. Ad esempio in listato 2.1 il metodo "next" è invocato su un'espressione di tipo "FixNum".

Listato 2.1: Esecuzione di un'espressione in IRB, Interactive Ruby Shell

```
1 2.1.1.1 :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 2.2: 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
```

Come si può vedere in listato 2.2 la dichiarazione di una classe incomincia con la keyword “class” seguita 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 con “@@”.

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 2.3: Modifica 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 2.3 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<sup>1</sup> per l'accesso all'attributo pubblico “color” mantenendo anche le precedenti funzionalità, come il metodo “area”.

<sup>1</sup>Il metodo “puts” stampa su standard output.

L'elemento `:"color"`, utilizzato per creare i metodi d'accesso alla colorazione del rettangolo, è un simbolo~[3]. A differenza delle stringhe~[4], 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  $O(1)$ .<sup>2</sup>

Listato 2.4: 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"`~[6], 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.~[7]

## Ereditarietà

Listato 2.5: 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"`~[8] che a sua volta estende `"BasicObject"`~[9], la radice della gerarchia.

Listato 2.6: Navigazione della gerarchia delle classi.

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

<sup>2</sup>Dalla prossima versione di Ruby~[5], la 2.2, sarà introdotto il symbol garbage collector. Al momento i simboli generati non rilasciano mai la memoria allocata introducendo potenziali memory leak.



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

---

Listato 2.7: Definizione di un nuovo modulo.

---

```
1 module Greeter
2   def greet salute = 'Hi'
3     puts salute
4   end
5 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 2.8: Mixin dei moduli “Greeter” e “EnthusisticGreeter”, per convenzione le costanti sono dichiarate utilizzando lettere maiuscole.

---

```
1 module EnthusisticGreeter
2   include Greeter
3   HYPE = 2
4   def greet_with_enthusiasm
5     HYPE.times do
6       greet 'Hello!!!'
7     end
8   end
9 end
10
11 class Person
12   include Greeter
13   include EnthusisticGreeter
14 end
15
16 bob = Person.new
17 bob.greet
18 => Hi
19 bob.greet_with_enthusiasm
20 => Hello!!!
21 => Hello!!!
```

---

## I blocchi

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

---

Listato 2.9: Le sintassi disponibili per i blocchi.

---

```
1 array = [1,2,3,4,5]
2 array.each {|i| puts i}
3 array.each do |i|
4   puts i
5 end
6 => 1
```

```
7 => 2
8 => ...
```

---

I blocchi, posizionati dopo l'ultimo parametro del metodo, possono essere definiti da una coppia di parentesi graffe (riga 2 in listato 2.9) oppure con la sintassi “do ... end” (riga 3 in listato 2.9), 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 2.10: 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~[10], “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~[11] 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.

### 2.1.2 RubyMine

Per lo sviluppo di RBlog e la definizione dei test di accettazione è stato utilizzato RubyMine~[12], 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 raramente sono stati rilevati 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<sup>3</sup> 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.

---

<sup>3</sup>Sono inclusi diversi plugin per il supporto sistemi di versionamento: attualmente sono disponibili CVS, Git, Subversion, Mercurial e Perforce.

## 2.2 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 degli 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.

### 2.2.1 Il modello

L'interpretazione di RoR del modello dell'architettura MVC prende spunto dal pattern Active Record~[13] 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 di interrogazioni attraverso una libreria di sistema.

Listato 2.11: Esecuzione di una query in Ruby che restituisce tutti gli articoli esistenti, dal più recente al più vecchio.

---

```
1 @posts = Post.order('updated_at DESC')
```

---

### 2.2.2 I controlli

Un controllo in RoR~[14] è rappresentato da una classe che estende "ActionController::Base"~[15] e fornisce dei metodi pubblici a cui corrispondono le richieste soddisfacenti dall'applicazione.

Listato 2.12: Frammento del controllo dei Post.

---

```
1 class PostsController < ApplicationController
2   # GET /posts
3   # GET /posts.json
4   def index
5     if params[:search].present?
6       @posts = Post
7         .where('title like ?', "#{params[:search]}%")
8         .order('created_at DESC')
```

---

```
9      else
10        @posts = Post.all.order('created_at DESC')
11      end
12    end
13  end
```

---

Nel listato 2.12 è visibile il controller<sup>4</sup> 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.

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*”~[16], 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 2.13: Frammento del file “*routes.rb*” relativo ai controlli di post e sessioni.

---

```
1 resources :posts do
2   get :autocomplete_title, :on => :collection
3 end
4 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 utilizzate per la gestione dei meccanismi di autenticazione.

Un altro accorgimento apprezzato durante lo sviluppo è la possibilità di definire dei filtri~[17] per i controlli. Per ogni azione dei controlli sono previsti alcuni stati<sup>5</sup> a cui è possibile attribuire delle callback. Sfruttando questa semplice tecnica è possibile fattorizzare alcuni comportamenti comuni all’interno di un controllo.

Listato 2.14: 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 2.15: 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
```

---

<sup>4</sup>PostController estende ActionController::Base tramite la classe ApplicationController.

<sup>5</sup>Paragonabili alle annotazioni @After, @Before, @AfterClass e @BeforeClass in JUnit 4.

“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.

### 2.2.3 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”<sup>6</sup>. Un documento ERB estende le pagine HTML classiche, aggiungendo la possibilità di scrivere codice Ruby per definire dei comportamenti dinamici.

Listato 2.16: Hello ERB!

```
1 <html>
2   <body>
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 2.17: 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~[19] 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.

<sup>6</sup>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”.

Le viste scritte tramite ERB sono facilmente leggibili grazie alle caratteristiche di Ruby<sup>7</sup> e alle numerose funzionalità presenti negli Helper e nelle librerie del linguaggio.

Listato 2.18: 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”~[21], che 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.

## 2.2.4 Il testing

Tramite RoR è possibile gestire l'intero stack di un'applicazione web, test inclusi~[22]. Sfruttando il framework RSpec~[23], 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~[24], 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~[25] è molto diffusa per la pulizia dei database utilizzati, così come Factory Girl~[26] per la popolazione del modello.

---

<sup>7</sup>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' 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~[27] per poter gestire al meglio l'esecuzione selettiva delle proprie librerie di test.

### 2.2.5 Peculiarità

L'obiettivo 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 utilizza RubyGems~[28], un package manager simile a Maven~[29] e Gradle~[30] 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~[31], 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 2.19: Frammento del Gemfile di RBlog.

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

Fra le gemme utilizzate, la più utile è stata Spring~[32]. 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~[33].



Listato 2.20: Migrazione relativa all'introduzione dell'entità Post nel modello.

```
1 class CreatePosts < ActiveRecord::Migration
2   def change
3     create_table :posts do |t|
4       t.string :title
5       t.text :body
6       t.timestamps
7     end
8   end
9 end
```

In RoR è previsto il meccanismo delle migrazioni~[34] 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.; gli identificatori 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 sviluppatori 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"~[35] dei parametri.

Listato 2.21: 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 2.21 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 2.22: 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, che permettono la definizione di viste come nella figura 2.1.

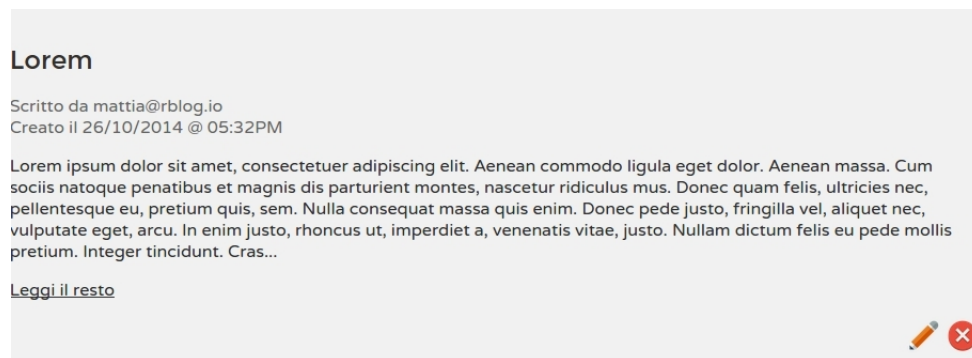


Figura 2.1: Visualizzazione di un singolo post in RBlog.

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 2.23: 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 2.24: 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~[36], un formato di serializzazione facilmente leggibile e scrivibile, e Ruby stesso.

Il listato 2.25 descrive la configurazione della connessione ai database PostgreSQL~[37] 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 2.25: 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.

## 2.3 Hello RBlog!

L'obiettivo 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.<sup>8</sup>

### 2.3.1 Cucumber

Il primo tassello scelto per l'implementazione dei test di accettazione su RBlog è Cucumber~[38].

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

Listato 2.26: La prima feature di RBlog

---

**Le funzionalità**

```

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 obiettivo 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'obiettivo 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~[39] secondo il formato "*Per* <beneficio>, *come* <ruolo>, *vorrei* <obiettivo, 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'estensione "*.feature*".

Nella funzionalità 2.26, 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 caratterizzano. L'etichetta è stata introdotta per poter eseguire in maniera selettiva gli scenari della prima funzionalità.

---

<sup>8</sup>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 2.27: 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 2.28: 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 2.26 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 "Dato", "Quando" e "Allora".

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 influenza l'esecuzione dello scenario, 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 2.29: 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 2.29 è eseguito sia sulla pagina dell'autore che sulla pagina dell'abstract.

---

Listato 2.30: Testo generato dallo scenario 2.29

---

```

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

```

---

**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.

### 2.3.2 Capybara

Capybara~[42] 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 2.31: 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. Tutti i driver~[43] devono implementare le funzionalità obbligatorie indicate nella libreria, ma è consentito non fornire il resto delle operazioni ed essere comunque annoverati fra i driver esistenti per Capybara. Per RBlog è stato utilizzato PhantomJS~[44] ed il driver Poltergeist~[45], che verranno trattati in seguito.

Listato 2.32: 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~[46];
- `: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~[47], lo stesso utilizzato da Chrome e Safari.

E' particolarmente adatto all'esecuzione automatica di applicazioni web non richiedendo un framework per la gestione di GUI, funzionalità spesso mancante sui server che effettuano l'integrazione continua.<sup>9</sup>

## Poltergeist

Poltergeist<sup>10</sup> è un driver per il browser PhantomJS ed implementa la totalità delle funzionalità obbligatorie e molte delle opzionali disponibili. Dopo aver configurato

<sup>9</sup>PhantomJS è installabile tramite i pacchetti d'installazione presenti sul sito ufficiale <http://phantomjs.org/download.html>.

<sup>10</sup>L'installazione di Poltergeist è effettuata attraverso il gemfile e la gemma "poltergeist".

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-[48], 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. E' 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.<sup>11</sup>

### 2.3.3 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.

Listato 2.33: 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 di 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 2.34: Ipotetica implementazione ulteriore.

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

L'introduzione di un'ulteriore implementazione, anche se con tipo differente, genera un'ambiguità che secondo il comportamento standard di Cucumber non è risolta.<sup>12</sup> 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

<sup>11</sup>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.

<sup>12</sup>Sfruttando l'opzione `-guess` di Cucumber è possibile far variare il comportamento per la scelta dell'implementazione da applicare.



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~[49] 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;
- 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”~[50].

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.<sup>13</sup>

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.

---

<sup>13</sup>Il metodo `click_link` permette di specificare l'opzione `:href` per verificare l'uguaglianza del attributo href prima di effettuare il click sul collegamento.

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 2.33 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<sup>14</sup>, che può essere sia relativo che assoluto.

Listato 2.35: 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
```

---

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 HTML "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 2.35.

A differenza del passo 2.33, dove non sono presenti parametri, nel passo 2.35 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~[51].

**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~[52] fornisce la struttura per la definizione di funzionalità e scenari per il BDD;

---

<sup>14</sup>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.

- RSpec-Expectations~[53] è una libreria di metodi per definire asserzioni;
- RSpec-Mocks~[54] è un framework per l'implementazione di stub, oggetti mock, verifiche sull'invocazione di metodi e dell'interazione fra oggetti;
- RSpec-Rails~[55] 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`~[56] in Rails, oppure con le asserzioni definite in `MiniTest::Spec`~[57].

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.<sup>15</sup>

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 2.36: 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 2.37: 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 2.29 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 2.38: Implementazione di un'asserzione parametrica.

---

```

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

```

---

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

Nello scenario 2.29 è richiesto il confronto di un parametro, definito attraverso l'espressione regolare, ed il titolo della pagina.<sup>16</sup> 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 2.39: 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
```

---

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

---

<sup>16</sup>Capybara::DSL::page è un getter e restituisce la rappresentazione della pagina attualmente aperta nel browser.

## 2.4 Introduzione del CSS

Listato 2.40: 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.

### 2.4.1 “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-[58], 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 2.41: 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, 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.

### 2.4.2 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 2.42: 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 2.43: 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 2.44: 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 stata 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 2.45: 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. Le funzionalità mancanti di Capybara per ottenere i valori stilistici di un elemento del DOM, obbligherebbe l'uso di numerosi script JQuery. Utilizzando direttamente l'identificatore dell'elemento HTML, oltre a richiedere di aggiungere molti attributi "id" e "class" appositamente per l'implementazione dei test, non verifica la visibilità del tag all'interno della pagina. E' quindi necessario che lo sviluppatore effettui manualmente ulteriori controlli, replicando il comportamento di Capybara per i "find\_\*".

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. Esistono anche attributi, come "background-color" che non sono ereditari, nell'esempio è verificato che l'assenza di colore coincida con la trasparenza.

Listato 2.46: Precondizione sul colore dei collegamenti nell'intestazione.

---

```

1 Given(/^i collegamenti non hanno sfondo$/) do
2   @textual_header_link_divs.each do |banner_link_div|
3     id = banner_link_div[:id]
4     background_color = steps_helper.background_color("#{id}", page
5   )
6     expect(background_color).to eq('rgba(0, 0, 0, 0)')
7   end
8 end

```

---

Nell'implementazione del passo si può osservare come ogni collegamento nell'intestazione sia trasparente. La codifica rgba codifica attraverso i primi tre valori i colori rosso, verde e blu, mentre il quarto campo "alpha" può assumere valori fra 1.0, che rappresenta un colore completamente opaco, e 0.0, che coincide con la piena trasparenza.

Sia lo scenario che è stato descritto che quelli non trattati della funzionalità 4.23 non verificano proprietà elaborate dei fogli di stile, perché 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.

### 2.4.3 Il contesto degli scenari

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

Listato 2.47: Background della funzionalità 4.23.

---

```

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.<sup>17</sup>

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

---

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



### 2.4.4 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 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 breakpoint, 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.

### 2.4.5 XPath

Le funzionalità del modulo Finders permettono di effettuare query tramite XPath~[59].

Listato 2.48: Query per la selezione del primo elemento del "body".

```
1 Then(/^1'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 2.49: Query per la selezione dei "div" con un titolo corrispondente al parametro.

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

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".

## 2.5 Definizione del modello

Listato 2.50: 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'obiettivo 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.

### 2.5.1 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 stakeholders, 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 delle condizioni d'esecuzione, introducendo ad esempio l'esecuzione in parallelo di più scenari.

Gli scenari della funzionalità 2.50 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 2.51: 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~[60] in Cucumber sono state definite delle procedure per annullare tutte le modifiche effettuate sull'applicazione.<sup>18</sup>

Listato 2.52: 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 effettuare delle misurazioni sulla velocità degli scenari. 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 sulla verifica degli scenari.

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

### 2.5.2 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 disponibile 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
```

---

## 2.6 Login & Autorizzazione

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

Listato 2.53: Descrizione della funzionalità di autenticazione.

---

```

1 Funzionalità: Autenticazione su RBlog
2   Come Autore di RBlog
3   Vorrei che alcune operazioni sensibili fossero 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 stata 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[61].

Listato 2.54: Frammento del controllo per l'autenticazione.

---

```

1 class SessionController < ApplicationController
2   #..
3   def create
4     author = Author.authenticate(params[:email], params[:hpassword])
5     if author
6       session[:author_id] = author.id
7       redirect_to
8         :root, :notice => 'Login effettuato, benvenuto!'
9     else
10      redirect_to :log_in, :notice => 'Credenziali invalide.'
11    end
12  end
13  #..

```

---

Il controllo "SessionController" è stato aggiunto all'applicazione, e contiene le dichiarazioni delle azioni per le operazioni "new", "create" e "destroy", rispettivamente 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.

### 2.6.1 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 2.55: 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.<sup>19</sup>

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

Listato 2.56: 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(
8     page)
9   expect(pre_login_encrypted_session).not_to
10  eq(post_login_encrypted_session)
11 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.

## 2.6.2 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 2.57: Variazioni nella funzionalità di creazione dei post.

---

```

1 @cap3

```

---

<sup>19</sup>Sono presenti i metodi: "name", "value", "domain", "path", "secure?", "httponly?", "expires".

```

2 @clear_and_logout
3 Funzionalità: Gestione dei post
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.<sup>20</sup> 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 2.58: 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\_oro, @ogni\_notte-.

---

<sup>20</sup>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.

## 2.7 Asincronia

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

### 2.7.1 JavaScript

Listato 2.59: Introduzione di un breve script Javascript.

---

```

1 Funzionalità: Easter Egging
2   Come Sviluppatore
3   Vorrei che nel blog fosse presente un mio logo
4   Per firmare il mio lavoro

```

---

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

Listato 2.60: Footer di RBlog.

---

```

1 <div id="footer" onclick="switch_easter_egg()">
2   <p>© 2014 - Mattia</p>
3 </div>

```

---

Listato 2.61: Frammento della funzione per aggiunta e rimozione del logo.

---

```

1 function switch_easter_egg() {
2   var woodstock = $('#woodstock');
3   if (!woodstock.length) {
4     var img = document.createElement("img");
5     img.src = "/assets/woodstock.png";
6     img.id = "woodstock";
7     /*...*/
8     document.getElementById("footer").appendChild(img);
9   }else{
10    woodstock.remove();
11  }
12 }

```

---

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.

---

```

1 Given(/^non è presente il logo nell'intestazione$/) do
2   step 'è presente il piè di pagina'
3   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.

### 2.7.2 JQueryUI

JQuery UI~[62] 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 "li", 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.<sup>21</sup>

**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 2.62: 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          },  
11          failure: function () {  
12            console.log("Failure");  
13          }  
14        }  
15      }  
16    }  
17  }
```

<sup>21</sup>JQuery UI definisce anche diversi temi per i plugin, garantendo una migliore integrazione estetica.

```

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 2.63: Callback per la funzione “autocomplete”.

---

```

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

```

---

### 2.7.3 Scenari sull'auto-completamento

Rispetto alla funzionalità 2.59, 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 2.64: 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'obiettivo 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 potenzialmente 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 2.65: Utilizzo del metodo "synchronize".

---

```

1 def finished_all_ajax_requests?(page)
2   page.document.synchronize do
3     page.find('#ui-id-1')
4   end
5 end
6
7 def wait_for_ajax(page)
8   begin
9     Timeout.timeout(Capybara.default_wait_time)
10    do
11      loop until finished_all_ajax_requests?(page)
12    end
13  rescue Timeout::Error
14  end
15  yield if block_given?
16 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 rallentamenti nell'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.

---

```

1 Scenario: Autocompletamento della ricerca
2   Dato nell'intestazione è presente la barra di ricerca
3   Dato il post "Lorem Ipsum" esiste
4   Quando inserisco il testo "lor" da ricercare
5   Allora viene proposto il post "Lorem Ipsum"
6   Quando inserisco il testo "xyz" da ricercare
7   Allora non è proposto alcun post
8   ...

```

---

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 2.66: Compilazione del menù con autocompletamento.

---

```

1 When(/^inserisco il testo "([^"]*)" da ricercare$/) do |
    searched_text|
2   page.fill_in 'search', :with => searched_text
3   search_input = page.find('#search_input_text')
4   search_input.trigger(:focus)
5   #search_input.trigger(:keydown)
6   page.execute_script %Q{ $('#search_input_text').trigger('keydown
    ') }
7   steps_helper.wait_for_ajax page
8 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 2.67: Ricerca di un post tramite click di una delle opzioni proposte.

---

```

1 When(/^ricerco "([^"]*)"$/) do |searched_text|
2   step "inserisco il testo \"#{searched_text}\" da ricercare"
3   steps_helper.wait_for_ajax page do
4     regexp = Regexp.new(Regexp.escape(searched_text), 'i')
5     if page.has_css?('.ui-menu-item', :text => regexp)
6       post_hint = page.find(:xpath, "//li[@class = 'ui-menu-item']"
7       )
8       expect(post_hint.text).to match(%r{#{searched_text}}i)
9       post_hint.trigger(:mouseenter)
10      post_hint.click
11    end
12    find('#search_icon').click
13  end
14 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, 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 richieste accorgimenti.

## Capitolo 3

# Spring

### 3.1 Spring MVC

In questo capitolo è trattato lo sviluppo di SBlog con Spring, un framework per la definizione di applicazioni web scritto in Java, e l'implementazione della libreria di test di accettazione automatici utilizzata per RBlog con Cucumber JVM e Selenium Web Driver.

La struttura di questo capitolo su Spring è simile al precedente: le prime sezioni descrivono l'architettura del framework, introducendo gli strumenti più significati nella definizione del nostro caso di studio nell'ecosistema di Java, dalla sezione 3.3 è descritto il processo d'implementazione delle funzionalità e scenari già descritti nel precedente capitolo, documentando tramite esempi e frammenti di test lo sviluppo dei test di accettazione.

#### 3.1.1 Spring Tool Suite

Spring Tool Suite, STS, è una versione personalizzata di Eclipse per supportare lo sviluppo tramite il framework Spring. I plugin presenti nell'ambiente di sviluppo forniscono supporto per i diversi linguaggi nello sviluppo web, come HTML, CSS o Javascript, l'integrazione con le componenti del framework, come la creazione di progetti guidata o la rappresentazione grafica delle componenti dell'applicazione, e le funzionalità già presenti nella versione di Eclipse per Java EE~[73], Enterprise Edition.

Per lo sviluppo di SBlog è stata utilizzata la versione 3.6.2 di Spring Tool Suite~[74], basata su Eclipse Luna 4.4. Oltre ad utilizzare la versione completa disponibile sul sito di Spring, è possibile anche installare le funzionalità di STS in una versione già esistente di Eclipse tramite l'apposito plugin.

## 3.2 L'interpretazione di Spring del pattern MVC

Nella sezione saranno descritti le componenti dell'architettura MVC in funzione dell'interpretazione data da Spring. Rispetto a RoR, come descritto in seguito, il framework Java necessita di una maggiore configurazione delle sue componenti, sono quindi inclusi numerosi esempi per dare un'indicazione del processo di sviluppo e alcuni riferimenti.

### 3.2.1 Il modello

Per l'implementazione del modello in Spring è stato utilizzato JPA~[75]<sup>1</sup>, Java Persistence API, una specifica Java per l'accesso, la persistenza e la gestione dei dati di DBMS relazionali, nella quale sono descritti i criteri per mappare un POJO, Plain Old Java Object, su un database.

Il vantaggio nell'utilizzo di una libreria che implementi la specifica, JPA infatti non fornisce alcuna implementazione, consiste nel poter modellare il dominio attraverso classi Java. Un POJO, termine introdotto da Martin Fowler~[63], è un oggetto Java che rappresenta le informazioni di un'entità del dominio e le relazioni che intercorrono con altri elementi attraverso l'insieme dei suoi attributi.

EclipseLink~[77] è una delle implementazioni esistenti per JPA, fornisce tutte le funzionalità previste dallo standard alla versione 2.1, e rispetto ad altre librerie come Hibernate, attualmente è leggermente più performante ed ha una miglior gestione del caricamento delle entità.

JPA 2.1, la versione attuale, fornisce agli sviluppatori sia strumenti per effettuare la definizione dello schema sia per effettuare interrogazioni senza utilizzare SQL puro, mantenendo parte dell'astrazione introdotta utilizzando la tecnica ORM.

#### DDL con JPA

Lo schema del modello di SBlog è stato definito esclusivamente tramite le annotazioni previste dalla specifica, JPA fa ampio uso del meccanismo della reflection<sup>2</sup> per analizzare a run-time le entità esistenti.

Il dominio di SBlog è composto da due entità "Author" e "Post" che rispettivamente descrivono gli autori del blog e gli articoli presenti, tramite i propri attributi d'istanza.

Listato 3.1: Dichiarazione dell'entità Post.

```
1 @Entity
2 public class Post {
3     /*...*/
```

<sup>1</sup>Rispetto a RoR, in Spring si gode di estrema libertà nella scelta delle librerie da utilizzare per la propria applicazione web.

<sup>2</sup>Il meccanismo della reflection in Java permette di accedere durante l'esecuzione, ma non di modificare, a codice non necessariamente noto al momento della compilazione e di compiere un'ispezione al fine individuare elementi di interesse.

---

```
4 }
```

Per dichiarare una nuova entità è necessario definire una classe Java e utilizzare l'annotazione “@Entity”, che presenta diversi overload per specificare parametri come il nome della tabella utilizzato nello schema.

---

Listato 3.2: Dichiarazione di alcuni attributi per l'entità Post.

---

```
1 @Id
2 @GeneratedValue(strategy = GenerationType.TABLE)
3 Integer id;
4
5 @Column(nullable = false, unique = true)
6 String title;
```

---

Rappresentata la tabella, per aggiungere delle colonne è sufficiente introdurre degli attributi d'istanza ed annotarli con “@Column” o altre annotazioni più specifiche, come nell'esempio in listato 3.2.

Tramite i parametri delle annotazioni è possibile dichiarare dei vincoli sui valori, ad esempio il titolo di un post deve essere sempre presente ed essere unico. Il tipo della colonna è inferito dal tipo Java, il tipo “Integer” corrisponde a “integer” in PostgreSQL e “String” a “varchar(255)”.

Per la definizione di colonne che rappresentano la chiave primaria della tabella è stata utilizzata l'annotazione “@Id” generando i valori in maniera univoca all'interno della tabella corrispondente all'entità “Post”.

---

Listato 3.3: Dichiarazione di un attributo temporale.

---

```
1 @Temporal(TemporalType.TIMESTAMP)
2 Date updatedAt;
```

---

L'annotazione “@Temporal”<sup>3</sup> permette di rappresentare sul modello attributi di tipo “Date” e “Calendar”, non è ancora possibile utilizzare la classe “LocalDate”~[76] introdotta con Java 8 che fornisce diversi metodi per una migliore gestione delle fasce orarie.

Le relazioni nell'approccio POJO sono rappresentate indirettamente attraverso attributi corrispondenti alle proprietà di navigazione nelle classi interessate dalla relazione. Ad esempio in listato 3.4 l'attributo “posts” della classe “Author” associa a ciascun autore i suoi post e, allo stesso tempo, l'attributo “author” della classe “Post” associa a ciascun post il suo unico autore, rappresentando così una relazione 1-N navigabile in entrambe i sensi.

Scegliendo di rappresentare la relazione con un oggetto “Set” è possibile gestire la cardinalità dell'associazione direttamente al livello d'astrazione dei POJO.

In JPA è possibile dichiarare relazioni unidirezionali, in cui l'associazione è mantenuta da un solo oggetto e non è possibile effettuare la navigazione in direzione

---

<sup>3</sup>L'enumerazione “TemporalType” permette di specificare il tipo da mantenere sul database, scegliendo fra “java.sql.Date”, “java.sql.Time” o “java.sql.Timestamp”.



contraria, oppure bidirezionali.

Oltre agli attributi primitivi che rappresentano le colonne delle tuple, sono presenti attributi composti per rappresentare le relazioni: ogni articolo ha un solo autore ed ogni autore può scrivere più articoli. In listato 3.4 si presenta un esempio di definizione di relazione uno a molti.

Listato 3.4: Le relazioni presenti fra le entità del modello.

---

```

1 public class Author {
2     /*...*/
3     @OneToMany(mappedBy = "author", ..., fetch = FetchType.LAZY)
4     Set<Post> posts;
5 }
6 public class Post {
7     /*...*/
8     @ManyToOne(..., fetch=FetchType.LAZY)
9     Author author;
10 }
```

---

Il parametro “fetch” nelle annotazioni che rappresentano le relazioni, è possibile caricare in maniera “lazy” le entità correlate, come i post scritti da un autore, posticipando l'operazione al momento dell'eventuale accesso, occupando così meno memoria. Sfruttando il caricamento degli oggetti in maniera “eager”, EclipseLink carica dal modello ogni attributo direttamente o indirettamente interessato dalla query, evitando successive interazioni con i database, per migliorare le performance.

Listato 3.5: Attributo posts per la classe Author.

---

```

1 @OneToMany(mappedBy = "author",
2     cascade = { CascadeType.ALL },
3     fetch = FetchType.LAZY)
4 Set<Post> posts;
```

---

L'attributo “cascade” definisce i vincoli di integrità referenziale di propagazione. A differenza di altre proprietà del DDL di JPA, la propagazione delle operazioni non ha ripercussioni sullo schema dei database ed è gestita direttamente sui POJO. In tal modo si possono usare anche database che non supportano i vincoli di integrità, come MySQL~[78]senza InnoDB~[79].

La mancanza di controlli sul DBMS obbliga a descrivere accuratamente la logica delle proprie operazioni CRUD, per mantenere la consistenza fra i record dei database e gli oggetti ed evitare errori.

Oltre alle relazioni utilizzate per SBlog sono ovviamente disponibili le annotazioni per implementare associazioni con altre cardinalità.

Listato 3.6: Attributo author per la classe Post.

---

```

1 @ManyToOne(cascade = { CascadeType.MERGE, CascadeType.REFRESH },
2     fetch=FetchType.LAZY)
3 @JoinTable(name = AUTHOR_POST_JOIN_TABLE,
```

---

```

4      joinColumns = @JoinColumn(name = POST_JOIN_COLUMN),
        inverseJoinColumns = @JoinColumn(name = AUTHOR_JOIN_COLUMN),
        uniqueConstraints = @UniqueConstraint(columnNames
        = {
            AUTHOR_JOIN_COLUMN,
            POST_JOIN_COLUMN}))
5  Author author;

```

In listato 3.6, l'annotazione “@JoinTable” dell'attributo “author” specifica la tabella ausiliaria in cui mantenere la relazione, che conterrà coppie formate da identificatori dell'autore e dell'articolo. Nelle tabelle ausiliaria ciascuna coppia comparirà un'unica volta, grazie all'annotazione “@UniqueConstraint” che introduce un vincolo di unicità.

### Generazione dello schema

La configurazione di Spring avviene principalmente tramite DI, Dependency Injection, e la definizione di bean. I bean sono metodi factory che istanziano oggetti rappresentanti le diverse proprietà configurabili del framework; tramite la DI gli oggetti sono istanziati ed utilizzati quando opportuno.<sup>4</sup>

Listato 3.7: Configurazione del metodo factory per la gestione delle entità del modello.

```

1  @Bean
2  public LocalContainerEntityManagerFactoryBean entityManagerFactory
3      () {
4      LocalContainerEntityManagerFactoryBean em = new
5          LocalContainerEntityManagerFactoryBean();
6      em.setDataSource(dataSource());
7      em.setPackagesToScan("sblog.orm");
8      JpaVendorAdapter vendorAdapter = new EclipseLinkJpaVendorAdapter
9          (); em.setJpaVendorAdapter(vendorAdapter);
10     em.setJpaProperties(additionalProperties());
11     em.setJpaDialect(new EclipseLinkJpaDialect());
12     return em;
13 }

```

Per impostare la connessione ai database e configurare l'entity manager è necessario introdurre almeno due bean. In particolare il metodo factory 3.7 specifica l'implementazione scelta per JPA e alcune proprietà attraverso un apposito metodo “additionalProperties”.

Fra le proprietà configurabili di EclipseLink è possibile scegliere se generare direttamente il database o un script contenente il DDL, inoltre ad ogni avvio può essere verificato lo schema ed eventualmente apportate delle modifiche se sono stati fatti dei cambiamenti alle classi delle entità.

### Query

Per semplificare la definizione di operazioni CRUD e effettuare delle interrogazioni sul dominio, Spring include una gerarchia di interfacce Java generiche-[80].

<sup>4</sup>La definizione dei bean può avvenire sia tramite factory method opportunamente annotati, sia utilizzando file XML.

Listato 3.8: Interfaccia `PostRepository` per la gestione e interrogazione dell'entità `Post` sul modello.

```
1 public interface PostRepository extends JpaRepository<Post, Integer> {  
2     > {  
3         public Post findPostByTitle(String title);  
4         public List<Post> findPostByTitleContainingIgnoreCase(String  
            title);  
5     }  
6 }
```

Per utilizzare le funzionalità presenti nella libreria è stata creata un'interfaccia "PostRepository" che estende "JpaRepository". Come si può osservare nell'implementazione, l'interfaccia padre è generica sulla classe che rappresenta i post ed il tipo della chiave primaria.<sup>5</sup>

L'interfaccia "JpaRepository" estende le interfacce generiche "CrudRepository", che fornisce i metodi corrispondenti alle operazioni CRUD, "PagingAndSortingRepository" che definisce la possibilità di effettuare interrogazioni paginate<sup>6</sup> e "Repository" che è la radice della gerarchia.

L'implementazione delle funzionalità dichiarate nelle interfacce è fornita dalla libreria JPA Criteria~[82], che inferisce dinamicamente dalla segnatura del metodo Java una query JPQL~[81], Java Persistence query language.

JPQL è un linguaggio per la definizione di query che utilizza il paradigma ad oggetti, ad esempio le proiezioni SQL sono effettuate utilizzando la notazione puntata per l'accesso agli attributi.

E' possibile estendere le numerose funzionalità presenti nella libreria dichiarando nuovi metodi, nell'interfaccia 3.8 sono stati introdotti due metodi che rispettano le convenzioni introdotte dalla libreria Spring Data Repositories e generano interrogazioni sulle entità `Post`: il metodo "findPostByTitle" individua un singolo oggetto in funzione del parametro, mentre "findPostByTitleContainingIgnoreCase" interroga il modello per ottenere le entità il cui titolo soddisfa l'espressione regolare.

Come si può intuire dal secondo metodo, l'implementazione di interrogazioni con un alto numero di condizioni di selezione comporta la dichiarazione di metodi con signature eccessivamente lunghe e complicate, inoltre non è possibile utilizzare l'operazione di join fra entità. Per evitare a questi problema è possibile annotare i metodi del interfaccia repository con "@Query" specificando direttamente il testo dell'interrogazione in JPQL.

L'interfaccia "JPAREpository" non rappresenta la scelta migliore per effettuare operazioni di selezione, ma è comunque di grande utilità per lo sviluppatore perché definisce anche tutti i metodi necessari per compiere operazioni di manipolazione del modello (create, delete, update).

<sup>5</sup>L'uso di "Integer" come tipo della chiave primaria dell'entità `Post`, piuttosto che utilizzare il tipo primitivo "int", è stato dettato dalla necessità di usare il tipo come parametro generico di "JpaRepository".

<sup>6</sup>Data un'interrogazione, la paginazione restituisce un sottoinsieme delle entità risultato in funzione dei parametri di cardinalità e ordinalità, rispettivamente il numero di elementi del risultato e la pagina scelta.

### 3.2.2 I controlli

Rispetto a RoR, la cui configurazione dei controlli è principalmente inferita in funzione delle convenzioni del framework, in Spring è necessario utilizzare un discreto numero di annotazioni.

Come per RBlog, sono stati sviluppati tre controller per gestire le richieste riguardanti gli autori, i post e le procedure di autenticazione ed autorizzazione. Un controller in Spring coincide con una classe, annotata con “@Controller” che ne definisce il ruolo.

Listato 3.9: Dichiarazione di un controller per i Post.

---

```

1 @Controller
2 @RequestMapping(value = "/posts")
3 public class PostController extends AbstractController {
4     /*...*/
5     @RequestMapping(value =("/{id})", method = RequestMethod.GET)
6     public String show(@PathVariable Integer id, Model model) {
7         Post post = /*...*/
8         model.addAttribute("page_title", post.getTitle());
9         model.addAttribute("content_template", "/posts/show");
10        model.addAttribute("posts", new Post[] { post });
11        return super.defaultMapping(model);
12    }
13    /*...*/
14 }
```

---

“PostController” è la classe che gestisce le richieste riguardanti gli articoli presenti sul blog. L’annotazione “@RequestMapping” definisce i parametri delle richieste d’instradamento ed è utilizzabile sia per classi che singoli metodi. L’uso dell’annotazione per la classe specifica la radice delle richieste soddisfacenti dai metodi presenti.

Il metodo “show”, che permette la lettura di un singolo post, soddisfa l’instradamento delle richieste HTTP del tipo “posts/42” concatenando gli attributi “value” dell’annotazione sulla classe e sul singolo metodo, semplificando la definizione di controlli in stile REST.

Oltre a specificare gli URL delle richieste, tramite l’annotazione “@RequestMapping” è possibile configurare quali tipi di connessioni HTTP sono accettati dal metodo, “show” ad esempio accetta esclusivamente richieste in GET.

Una particolarità del metodo “show” è di essere parametrico sull’identificatore intero dei post, l’attributo “value” “/{id}” associa, previa conversione di tipo, il valore della stringa all’oggetto “id” di tipo “Integer” annotato tramite “@PathVariable”.

La struttura delle azioni a cui è associata una pagina web, come il metodo “show”, prevede un oggetto di tipo “Model”, utilizzato come parametro di output, al quale aggiungere coppie chiave-valore rappresentanti gli attributi dinamici necessari alla vista e una stringa come valore di ritorno per specificare l’elemento da renderizzare.

Per rendere più modulare SBlog, il ritorno delle azioni restituisce sempre una stringa che fa riferimento al layout del sito. Il parametro “content\_template” utilizzato da un'apposita espressione in Thymeleaf, il template engine scelto per SBlog, definisce il contenuto della pagina, nell'esempio verrà visualizzata la vista “/posts/-show”. Maggiori dettagli su Thymeleaf sono presenti nella successiva sezione.

Listato 3.10: Implementazione di un metodo privo di interfaccia web.

---

```

1 @RequestMapping(value = "/autocomplete_title", method =
    RequestMethod.GET)
2 @ResponseBody
3 public String[] autocompleteTitle(@RequestParam(value = "title",
    required = true) String title) {
4     return postService.queryByTitle(title);
5 }

```

---

Per gestire le richieste HTTP di tipo GET, che includono i parametri direttamente nell'URL, è utilizzata l'annotazione “@RequestParam” che permette di ottenere un riferimento ai valori forniti e di introdurre anche dei vincoli, nell'esempio è richiesto che sia presente un parametro “title”. Oltre agli attributi “value” e “required” è anche possibile definire un valore di default da applicare in assenza del dato.

Per ottenere un riferimento ad un parametro presente nel body di una richiesta HTTP è possibile utilizzare l'annotazione “@RequestBody”.

La particolarità del metodo “autocompleteTitle” rispetto all'azione “show” descritta in precedenza è nel tipo di risposta che fornisce. Il metodo, definito per l'auto-completamento del widget di JQuery UI già presente in RBlog, restituisce la serializzazione di un array di stringhe invece che il riferimento ad una vista. L'annotazione “@ResponseBody” indica che il valore di ritorno dev'essere inserito all'interno di un “body” di una risposta web.

Le azioni di Spring utilizzano ampiamente i parametri come output. Introducendo ulteriori parametri rispetto a quelli descritti finora negli esempi è possibile attivare e gestire funzionalità utili per le viste.

---

```

1 @RequestMapping(value = "/new", method = RequestMethod.POST) public
    String newPost(@Valid Post post, BindingResult bindingResult,
        Model model, RedirectAttributes redirectAttributes,
        HttpSession httpSession) {
2     /*...*/
3 }

```

---

Ad esempio, la vista relativa alla scrittura di un nuovo articolo prevede la gestione di numerosi aspetti, a differenza di RoR che disaccoppia le azioni di gestione dei form e della persistenza dell'entità, in Spring i due comportamenti sono implementati insieme. I parametri utilizzati sono:

- un'istanza di tipo “Post” che rappresenta l'articolo, l'oggetto può essere vuoto in quanto la pagina è appena stata visualizzata o contenente i dati dopo aver effettuato il salvataggio;

- un oggetto “BindingResult” che contiene i possibili messaggi di errore del processo di validazione per ogni elemento del form;
- un oggetto “Model” per fornire parametri alla vista;
- un oggetto “RedirectAttributes” per gestire la ridirezione in caso di salvataggio del post;
- un parametro “HttpSession” per la gestione della sessione e la verifica delle autorizzazioni;

La tecnica scelta da Spring per gestire i diversi aspetti comporta un'eccessiva mole di parametri e funzionalità da implementare in un singolo metodo. Sta allo sviluppatore intervenire fattorizzando le azioni e gestire manualmente le diverse casistiche, operazioni comodamente implementate di default in RoR alla creazione di un nuovo controllo.

### I Service

Per fattorizzare la logica dei controller Spring definisce i service, classi Java annotate con “@Service” che hanno il compito di far interagire i controller con le classi repository definite dall'utente.

In SBlog è presente un “SessionService”, che fornisce le funzionalità per l'autenticazione e la codifica delle password, e “PostService” che assicura la consistenza delle informazioni sul modello, ad esempio inizializzando la data di creazione e di aggiornamento un articolo alla creazione o alla modifica.

I service, così come gli helper in RoR, permettono di disaccoppiare la logica delle operazioni dalle azioni dei controller, permettendo un'implementazione più semplice delle componenti.

### La dependency injection

Per utilizzare al meglio le diverse componenti in SBlog è stato fatto ampio uso dell'annotazione “@AutoWired”.

Listato 3.11: Uso dell'annotazione @Autowired.

```
1 @Controller
2 @RequestMapping(value = "/posts")
3 public class PostController extends AbstractController {
4     @Autowired
5     PostService postService;
6     @Autowired
7     AuthorService authorService;
8     @Autowired
9     SessionService sessionService;
10    /*...*/
11 }
```

Per ogni componente esistente in Spring, sia essa un “@Repository”, “@Controller” o “@Service”, è generato un corrispondente “bean”. Grazie all'annotazione

“@AutoWired” e alla dependency injection, le corrette istanze dei componenti sono “iniettate” quando necessario, semplificando notevolmente la risoluzione delle dipendenze.

### 3.2.3 Le viste

Per l'implementazione delle viste di SBlog è stato utilizzato Thymeleaf~[83], una libreria Java per la generazione dinamica di documenti web. La libreria, organizzata in due moduli contenenti le funzionalità standard e quelle specifiche per Spring, permette la definizione di pagine dinamiche utilizzando esclusivamente elementi sintattici esistenti nell'HTML. Al contrario ERB e Razor, il template engine di MVC5 descritto nel prossimo capitolo, prevedono l'utilizzo di delimitatori per includere frammenti di codice non interpretabili da un browser ma da processare tramite un server web.

Dopo essermi documentato sui template engine comunemente utilizzati con Spring e Java, ho scelto di non implementare le viste tramite JSP~[84], JavaServer Page. Il motivo principale della scelta sono i diversi problemi tecnici nella configurazione della libreria utilizzando Spring MVC 4 e Spring Boot, che verrà discusso in seguito.

#### Thymeleaf

Tutti gli attributi utilizzati dalla libreria sono presenti nel namespace “http://www.thymeleaf.org”<sup>7</sup>. Una vista definita con Thymeleaf ha estensione e struttura identica ad un documento HTML statico: senza il supporto di un server web i tag introdotti dalla libreria non vengono analizzati dai browser permettendo l'implementazione di prototipi offline, per sviluppare l'interfaccia dell'applicazione non è quindi necessario aver già un'architettura MVC funzionante.

Listato 3.12: Dichiarazione del namespace.

---

```
1 <html xmlns:th="http://www.thymeleaf.org">
```

---

Come in RoR, uno degli obiettivi nella definizione delle viste è individuare delle tecniche per fattorizzare gli elementi comuni. Tramite l'attributo “th:replace” è possibile sostituire il nodo corrente con un'intera vista.

Listato 3.13: Struttura comune delle viste in SBlog.

---

```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3   <head th:replace="/layouts/head"/>
4   <body th:class="${class_name}">
5     <div id="header" th:replace="/layouts/header"></div>
6     <div th:replace="${content_template}"></div>
7     <div id="footer" th:replace="/layouts/footer"></div>
8   </body>
9 </html>
```

---

<sup>7</sup>In eclipse è presente un plugin per assistere lo sviluppatore nella definizione delle viste tramite l'auto-completamento e la colorazione delle parole chiave.

Le viste in SBlog prevedono un elemento “head”, contenente i riferimenti alle risorse come fogli di stile<sup>8</sup> e librerie JavaScript. Il “body” è suddiviso in tre principali aree: l'intestazione, il contenuto principale ed il piè di pagina. Indicativamente, ogni attributo presente in HTML5 ha una controparte in Thymeleaf per la definizione di comportamenti dinamici. Le funzionalità utilizzate ed alcuni frammenti delle viste che le contengono sono introdotte brevemente nelle prossime sezioni.

### Visualizzazione dei post

Listato 3.14: Iterazione su una collezione di post.

```
1 <div class='post' th:each='post : ${posts}'>
2   <p class="post_title">
3     /*...*/
4   </div>
5 </div>
```

L'attributo “each” presente nel primo nodo della vista parziale 3.14 permette di iterare su una collezione “posts” in maniera simile al costrutto “foreach” di Java. Gli oggetti presenti nella vista sono acceduti tramite l'istanza di “Model” presente nell'azione del controller che ha gestito la richiesta; Per ogni chiave presente nell'istanza di “Model”, l'espressione “\${key}” effettua l'accesso all'elemento associato alla chiave “key”.

Listato 3.15: Accesso agli attributi ed ai metodi di un oggetto nella vista.

```
1 <div class="post_detail" th:text="${post.creationDetail()}"></div>
2 <div class="post_detail" th:text="${post.author.authorDetail()}"></div>
```

All'interno delle espressioni di Thymeleaf gli attributi forniti alla vista sono oggetti ed è quindi possibile accedere ai metodi d'istanza e ai valori. L'attributo “text” sostituisce il corpo del nodo HTML con il valore dell'espressione, come in listato 3.15.

Listato 3.16: Utilizzo di un'espressione if.

```
1 <a th:if="${post.body.length() > 500}" th:href="/posts/${post.id}">
2   Leggi          il resto...
3 </a>
```

Oltre all'attributo “each” sono presenti altre istruzioni in listato 3.16, come “if” e “unless”, per controllare il flusso dell'esecuzione. Nell'esempio il collegamento verrà visualizzato in funzione della guardia, in generale l'uso di costrutti come “if” condiziona la visita da parte di Thymeleaf di tutto il sotto-albero.

<sup>8</sup>A differenza di RoR e RubyMine, nella definizione delle viste non è stato utilizzato Sass, o altre estensioni del CSS, per la mancanza di plugin in Eclipse che forniscano la compilazione automatica ed il supporto alla scrittura delle pagine di stile.



### Gestione dei form

La gestione dei form è uno dei contesti per cui Thymeleaf è stato specializzato per integrarsi al meglio con le funzionalità di Spring. Per implementare la verifica dei dati inseriti nei form, in Spring è possibile utilizzare le annotazioni presenti nel package “`javax.validation.constraints`”~[85] di JPA per validare esclusivamente i valori dei singoli attributi. Validazioni che coinvolgono più attributi, non sono però supportate, come verificare che il titolo di un post non sia già presente nel dominio.

A differenza di RoR, dove la validazione dei form coincide con la validazione effettuata dai database, gli strumenti di Spring replicano i meccanismi esistenti a basso livello e permettono di anticipare tali verifiche. Questa scelta comporta la duplicazione dei vincoli sul dominio, esprimendoli sia a livello logico che di persistenza, a favore di una maggiore efficienza di esecuzione.

Listato 3.17: Uso delle annotazioni di JPA per la validazione.

---

```

1 @Size(min=5, max=100, message = "Il titolo deve essere compreso fra
   5 e 100 caratteri.")
2 @NotNull(message = "Titolo mancante.")
3 @Column(nullable = false, unique = true)
4 String title;
```

---

Oltre alle annotazioni della libreria è possibile implementarne di personalizzate, ad esempio che verifichino l'unicità del titolo, definendo una nuova annotazione e una classe contenente la logica. Purtroppo nella definizione del tale vincolo, sono stati rilevati errori nell'uso dei repository istanziati tramite DI e ho dovuto implementarne la logica manualmente nel controllo dei post.

Gli errori del form sono forniti ai controller tramite un parametro di tipo “`BindingResult`”, già introdotto nella precedente sezione.

Listato 3.18: Frammento della vista per la visualizzazione degli errori sul campo del titolo.

---

```

1 <div class="error_explanation" th:if="${#fields.hasErrors('title')}>
   ">
2   <p id="error_description" th:errors="*{title}"></p>
3 </div>
```

---

All'interno delle viste di Thymeleaf sono state utilizzate le funzionalità della classe ausiliaria “`fields`”, che implementa le funzionalità per la gestione della compilazione dei form HTML.<sup>9</sup>

Rispetto a ERB e Razor, discussi in sezione 2.2.3 e 4.2.3, le viste sviluppate con Thymeleaf appaiono più semplici e leggibili. Utilizzando gli attributi per generare il contenuto dinamico si riduce sensibilmente la propensione ad introdurre logica all'interno delle viste, rispettando i ruoli delle componenti del pattern MVC. Tuttavia Thymeleaf introduce una sintassi articolata e non molto intuitiva che può costituire un ostacolo ad una rapida definizione della logica elementare di una vista, come la formattazione di una stringa o la generazione dinamica di collegamenti.

<sup>9</sup>Sono presenti altre classi per il supporto di operazioni comuni nelle viste, come la formattazione di date.

### 3.2.4 Il testing

Spring implementa le funzionalità per definire unit ed integration test. Per SBlog non sono stati implementati test oltre a quelli di accettazione, la struttura e la logica sono molto simili a quanto sviluppato su RoR, ma ritengo sia utile fornire qualche riferimento.

L'implementazione di unit-test in Spring è possibile grazie alla DI e a oggetti mock per poter testare in isolamento le singole funzionalità.

Per implementare i propri integration-test non è necessario effettuare il deploy dell'intera applicazione, infatti sfruttando le funzionalità presenti nel modulo "spring-test"~[64] è possibile verificare l'interazione delle diverse componenti. Tutti i test in Spring sono eseguibili tramite JUnit~[65]: l'annotazione "@RunWith"~[66] permette di configurare quale runner utilizzare, è quindi possibile uniformare l'ambiente di testing e le funzionalità di supporto per tutti i differenti livelli di verifica dell'applicazione. Maggiori dettagli su JUnit sono presenti nella prossima sezione.

### 3.2.5 Peculiarità

Spring è estremamente modulare e consente all'utente piena libertà nella scelta degli strumenti; il che comporta tuttavia la necessità di configurare i relativi bean. Per semplificare lo sviluppo dell'applicazione è stato utilizzato Spring Boot, un progetto degli stessi sviluppatori del framework.

Per ridurre al minimo la necessità di configurazione di una nuova applicazione, è possibile utilizzare i progetti "starter" esistenti.

Listato 3.19: Alcuni "starter" di Spring Boot utilizzati per SBlog.

---

```

1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>      <artifactId>
4       spring-boot-starter-web</artifactId>
5     </dependency>
6   <dependency>
7     <groupId>org.springframework.boot</groupId>
8     <artifactId>spring-boot-starter-thymeleaf</artifactId>
9   </dependency>
10  <dependency>
11    <groupId>org.springframework.boot</groupId>
12    <artifactId>spring-boot-starter-data-jpa</artifactId>
13    <exclusions>
14      <exclusion>          <artifactId>hibernate-entitymanager</
15        artifactId>
16      <groupId>org.hibernate</groupId>
17    </exclusion>
18  </exclusions>
19 </dependency>
20 /*...*/

```

---

Aggiungendo gli opportuni “starter” alle dipendenze del proprio progetto, (nell'esempio in listato 3.19 è mostrato un frammento del POM<sup>10</sup> di SBlog ma è possibile utilizzare anche Gradle), la maggior parte delle componenti e dei bean necessari sono configurati con le impostazioni più comuni. Ovviamente alcune proprietà specifiche, come le credenziali per la connessione ai database, sono da impostare manualmente.

Le componenti non configurabili tramite Spring Boot, sia perché troppo specifiche, sia perché riguardanti strumenti non presenti negli “starter”, sono configurabili tramite i bean. Spring, attualmente alla versione 4.1, è retro-compatibile permettendo la definizione di bean attraverso file XML, definendo una gerarchia di classi o tramite l'annotazione “@Bean” ed i factory method, tecnica utilizzata in SBlog.

---

<sup>10</sup>Un POM, Project Object Model, è un file XML utilizzato da Maven per la definizione delle informazioni del progetto. All'interno del file è possibile definire diversi processi di compilazione, ad esempio includendo o meno le librerie per il testing, dichiarare le dipendenze del progetto e molto altro.

### 3.3 Hello SBlog!

Per effettuare un confronto il più attendibile possibile, la libreria di test di accettazione definita per RBlog e scritta in Gherkin è stata riutilizzata anche per la versione del blog in Spring. I due blog sono equivalenti nelle funzionalità e molto simili nelle tecnologie utilizzate. In questa e nelle prossime sezioni è descritto il processo di definizione dei test, omettendo o semplificando la descrizione dei tool se utilizzati per RBlog.

#### 3.3.1 Cucumber-JVM

Cucumber-JVM~[86], il porting in Java del framework per Cucumber, presenta differenze minime rispetto alla versione per Ruby, principalmente dovute alle diversità del linguaggio.

**Le funzionalità** Nell'effettuare il porting dei test di accettazione in Gherkin non è stato rilevato alcun problema sintattico né relativo alle funzionalità supportate dalla versione Java di Cucumber.

Listato 3.20: La prima feature di SBlog

---

```

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

```

---

Avendo già descritto la struttura di una funzionalità in Gherkin, è possibile concentrare l'attenzione sulla configurazione del framework, presenti in listato 3.21.

Listato 3.21: Aggiunta delle dipendenze per il BDD in Maven.

---

```

1 <dependency>
2   <groupId>info.cukes</groupId>   <artifactId>cucumber-junit</
      artifactId>
3   <version>1.1.8</version>
4   <scope>test</scope>
5 </dependency>
6 <dependency>
7   <groupId>info.cukes</groupId>
8   <artifactId>cucumber-picocontainer</artifactId>
9   <version>1.1.8</version>
10  <scope>test</scope>
11 </dependency>

```

---

Per l'esecuzione dei test in Eclipse innanzitutto è necessario includere le dipendenze all'interno del POM di Maven. Oltre a "cucumber-junit" è necessario includere anche PicoContainer~[87], una libreria per il supporto alla DI utilizzata per individuare le classi contenenti gli hook e le definizioni dei passi.

Listato 3.22: Runner per Cucumber-JVM

---

```

1 @RunWith(Cucumber.class)

```

---

```

2 @CucumberOptions(format = {tags = {"@cap1", "~@ignore"}})
3 public class RunCukesTest {}

```

Aggiunte le dipendenze e le funzionalità, è sufficiente configurare un classe con le annotazioni di JUnit per eseguire i test. L'annotazione “@CucumberOptions” indica i tag degli scenari da eseguire, come illustrato in listato 3.22.

**Supporto a Cucumber in STS** In Eclipse è disponibile un plugin per Cucumber~[67] che fornisce la formattazione del codice scritto in Gherkin, l'auto-completamento e l'evidenziazione della sintassi per tutte le lingue supportate.

Rispetto all'integrazione in RubyMine manca la possibilità di navigare dal passo all'implementazione, inoltre la funzionalità per eseguire i test tramite le risorse scritte in Gherkin sembra non funzionare, obbligando ad eseguire direttamente il runner di JUnit.

### 3.3.2 Selenium

Selenium~[88] è una libreria per l'automazione la navigazione internet. Rispetto alle versioni precedenti della libreria, gli sviluppatori affermano di aver definito un'interfaccia più intuitiva e compatta.

Selenium WebDriver, la parte centrale della libreria che implementa le operazioni per l'accesso alle pagine web, utilizza il supporto nativo di ogni browser per l'automazione. La prima differenza rispetto a Capybara è nella possibilità utilizzare tutte le funzionalità di browser web, come la navigazione grazie alla cronologia e l'uso dei preferiti, piuttosto che limitarsi alla sola navigazione.

Listato 3.23: Aggiunta della dipendenza di Selenium a SBlog.

```

1 <dependency>
2   <groupId>org.seleniumhq.selenium</groupId> <artifactId>selenium-
      java</artifactId>
3   <version>2.43.1</version>
4 </dependency>

```

Per poter definire i test utilizzando le funzionalità di Selenium è sufficiente aggiungere la dipendenza del porting in Java al POM del progetto, come in listato 3.23.

### PhantomJS

Data la forte integrazione di Selenium con i browser web è stato scelto di adottare anche per i test su SBlog PhantomJS. Scegliendo di implementare i test di accettazione automatici con Cucumber, nelle varie versioni, ed utilizzando lo stesso software per la navigazione dei blog è più facile individuare per ogni problema tecnico la causan ed effettuare un'analisi più accurata.

### Ghost Driver

Ghost Driver~[89] è un'implementazione JavaScript del protocollo WebDriver Web di PhantomJS che permette di eseguire i comandi contattando tramite richieste HTTP l'interfaccia REST del browser, che restituisce i risultati delle interazioni serializzati con JSON.

Per configurare Ghost Driver è sufficiente aggiungere la dipendenza al POM, come in listato 3.24.

Listato 3.24: Aggiunta della dipendenza di Selenium a SBlog.

---

```

1 <dependency>
2   <groupId>com.github.detro.ghostdriver</groupId>
3   <artifactId>phantomjsdriver</artifactId>
4   <version>1.1.0</version>
5 </dependency>

```

---

### 3.3.3 Implementazione dei passi

L'implementazione di un passo in Cucumber-JVM è eseguita annotando un metodo pubblico<sup>11</sup> con una delle annotazioni corrispondenti ai passi e specificando il valore dell'espressione regolare. Inoltre è possibile specificare un parametro "timeout" per specificare il tempo massimo di esecuzione del passo.

Come per la versione in Ruby, non c'è corrispondenza fra il tipo dell'implementazione, in Java è rappresentato dall'annotazione, ed i tipi di passi applicabili.

Listato 3.25: Implementazione del passo "apro SBlog".

---

```

1 @Dato("^apro SBlog$")
2 public void apro_SBlog() {
3     /*...*/
4 }

```

---

Per implementare passi parametrici è sufficiente aggiungere dei parametri al metodo. Come in Ruby i valori estratti dall'espressione del passo hanno tipo stringa, ma è possibile annotare i parametri con "@Format" per effettuare eventuali conversioni.

Eventuali inconsistenze fra il numero di parametri del passo e del metodo sono segnalate tramite eccezione.

**La struttura di Selenium** Rispetto alla struttura di Capybara, Selenium organizza le funzionalità in maniera articolata~[68]. La libreria è rappresentata attraverso più gerarchie di classi, interfacce ed enumerazioni. Di seguito sono introdotti gli elementi più importanti e maggiormente utilizzati per il testing su SBlog:

- l'interfaccia "WebElement" rappresenta gli elementi HTML e include le principali operazioni eseguibili tramite Selenium. Le linee guida della libreria prevedono che ogni metodo effettui un controllo sulla validità del chiamante,

---

<sup>11</sup>Non è necessario configurare in maniera particolare le classi contenenti le implementazioni dei passi.

verificando che sia presente nel DOM al momento dell'esecuzione dell'operazione;

- l'interfaccia "WebDriver" è la radice della gerarchia delle operazioni che descrivono le operazioni eseguibili durante i test. Le funzionalità dichiarate nell'interfaccia riguardano il controllo del browser stesso, la selezione degli elementi presenti nel DOM e il supporto al debug durante i test;
- la classe astratta "By" implementa diversi factory method per individuare elementi all'interno del DOM. Il metodo "findElement" dell'interfaccia "WebElement" richiede un'istanza della classe "By" per compiere la ricerca nel documento. Rispetto a Capybara le possibilità per implementare un selettore sono indicativamente equivalenti, ma Selenium offre metodi come "ByTagName" o "ByPartialLinkText", derivati rispettivamente da "ByCssSelector" o "ByLinkText", che semplificano lo sviluppo dei test;
- l'interfaccia "WebDriverException" rappresenta la gerarchia degli errori presenti nella libreria.

**Navigare all'interno del sito** La navigazione in Selenium è effettuabile utilizzando tre tipologie di metodi.

Listato 3.26: Navigazione esplicita.

---

```
1 driver.navigate().to(getSblogURL());
```

---

- Il metodo "navigate", esemplificato in listato 3.26, restituisce un'istanza di tipo "Navigation" che permette al driver di accedere alle funzionalità del browser e di navigare, sia utilizzando la cronologia, sia indicando esplicitamente un URL da raggiungere. Il metodo "to" effettua richiesta in GET per l'URL specificato e blocca l'esecuzione fino al caricamento della nuova pagina.

Listato 3.27: Navigazione nel sito, sfruttando il testo visualizzato di un link.

---

```
1 WebElement link = driver.findElement(By.linkText(linkText));  
2 link.click();
```

---

- Oltre alla navigazione esplicita, è possibile compiere effettuare un'azione, come il click di un collegamento in listato 3.27, su un oggetto "WebElement" individuato tramite un selettore. Nell'esempio è effettuata la navigazione ad una delle pagine statiche del blog presenti per la prima funzionalità, individuando il link attraverso il testo degli elementi "a".
- Il metodo "findElement" restituisce il primo elemento che soddisfa il selettore di tipo "By". Per ogni ricerca all'interno del DOM è previsto un tempo massimo di attesa entro il quale l'elemento deve apparire ed una frequenza di polling. L'eccezione "NoSuchElementException" è sollevata nel caso l'elemento non sia presente.

**Integrazione con JUnit** Per l'esecuzione degli scenari e la definizione delle asserzioni è stato utilizzato JUnit, una libreria Java per la definizione di test. Le asserzioni in JUnit sono metodi statici della classe "Assert" ed offrono le funzionalità per la verifica dello stato di uno o più oggetti. Alcuni esempi sono riportati in listato 3.28 e 3.29.

Listato 3.28: Verifica del attributo "href" del collegamento presente nell'intestazione.

---

```
1 WebElement logo = driver.findElement(By.id("logo"));
2 assertEquals(logo.getAttribute("href"), getSblogURL());
```

---

Listato 3.29: Verifica del titolo della pagina.

---

```
1 assertEquals(driver.getTitle(), pageTitle);
```

---



## 3.4 Introduzione del CSS

Per semplificare lo sviluppo delle applicazioni è stato scelto di riutilizzare sia la struttura delle pagine che i fogli di stile creati per RBlog.

Listato 3.30: Seconda funzionalità per SBlog

---

```

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

```

---

Nonostante sia sempre più diffusa la pratica di utilizzare linguaggi come Sass, che arricchiscono CSS 3 come già discusso nel precedente capitolo, non è ancora fornito il supporto in Eclipse. Per evitare la riscrittura manuale ho compilato i fogli di stile tramite il comando “`sass *.scss`”<sup>12</sup> per convertirli al formato classico.

### 3.4.1 Testare il css

SBlog presenta quindi un layout molto simile alla precedente versione, compresi i minimi effetti cromatici dell'intestazione.

Listato 3.31: Variazione del colore di background dei collegamenti nell'intestazione.

---

```

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 stilistiche dei nodi** La funzionalità 4.23 richiede di verificare alcuni elementi del layout di SBlog. L'interfaccia “WebElement” dichiara il metodo “`getCssValue`” per ottenere stringhe rappresentanti gli attributi stilistici di un elemento del DOM.

Listato 3.32: Verifica del colore dello sfondo dei collegamenti.

---

```

1 @Dato("l'intestazione ha un colore di sfondo$")
2 public void l_intestazione_ha_un_colore_di_sfondo(){
3     String rgb = page.header.getCssValue("background-color");
4     assertNotNull(rgb);
5     assertEquals("rgba(0, 0, 0, 0)", rgb);
6 }

```

---

Come si può osservare dall'implementazione, Selenium rispetto a Capybara ottiene il corretto valore di un elemento del DOM, anche se il parametro è attribuito tramite un foglio di stile incluso nell'header della pagina, grazie ad una migliore integrazione con le funzionalità del browser.

<sup>12</sup>Il processo di conversione non gestisce al meglio i commenti presenti nei file scss, generando dei fogli di stile non validi. Per sicurezza infatti ho verificato i documenti con il validatore disponibile sul sito del W3C.

Purtroppo però, oltre alla classe “Color” che interpreta stringhe nel formato appena mostrato, non è presente nessuna funzionalità per semplificare l'utilizzo dei valori restituiti, lo sviluppatore deve quindi, tramite espressioni regolari e conversioni di tipo, analizzare manualmente le stringhe per effettuare operazioni più complesse.

Inoltre al metodo “getCssValue” è possibile fornire solo attributi validi per le specifiche del CSS 2, ad esempio l'attributo “margin” introdotto nel CSS 3, che attraverso una tupla di quattro valori definisce tutti i margini di un elemento HTML, non è supportato.

Listato 3.33: Asserzione sul colore di sfondo dell'intestazione e del piè di pagina.

```
1 @Allora("^intestazione e piè di pagina hanno lo stesso colore di
   sfondo$")
2 public void
   intestazione_e_piè_di_pagina_hanno_lo_stesso_colore_di_sfondo()
   {
3   String headerBackgroundColor, footerBackgroundColor;
4   headerBackgroundColor = page.header.getCssValue("background-color
   ");
5   footerBackgroundColor = page.footer.getCssValue("background-color
   ");
6   assertEquals(headerBackgroundColor, footerBackgroundColor);
7 }
```

Nonostante le difficoltà nell'utilizzo dei risultati, il metodo “getCssValue” risponde in maniera consistente ai principi del CSS. Il valore di default dell'attributo “background-color” per i collegamenti dell'intestazione coincide con la trasparenza massima, come mostrato nel prossimo frammento.

```
1 @Dato("^i collegamenti non hanno sfondo$") public void
   i_collegamenti_non_hanno_sfondo(){
2   List<WebElement> bannerLinks = page.header.findElements(By.
   className("banner_link"));
3   for (WebElement banner_link : bannerLinks) { String rgb =
   banner_link.getCssValue("background-color");
4   assertNotNull(rgb);
5   assertEquals("rgba(0, 0, 0, 0)", rgb); }
6   page.setBannerLinks(bannerLinks);
7 }
```

## Actions

Listato 3.34: Creazione di una macro azione.

```
1 @Quando("^il cursore si sposta sui collegamenti$")
2 public void il_cursore_si_sposta_sui_collegamenti() {
3   Actions action = new Actions(driver);
4   WebElement bannerLink = page.getBannerLinks().get(0);
5   action.moveToElement(bannerLink).perform();
6 }
```

La classe “Actions” permette di emulare complicate operazioni effettuate sul browser, definendo delle macro. E' possibile definire azioni di due tipi:

- eseguibili tramite tastiera, come la pressione di uno o più tasti combinati, ad esempio corrispondenti alle hotkeys implementate dal browser web;
- eseguibili tramite il mouse, come il semplice click o azioni più complesse come il drag & drop;

Nell'esempio in listato 3.34 è definita una macro azione<sup>13</sup> ed anche se descritta dal solo spostamento del cursore sopra l'elemento indicato, è utile per mostrarne il comportamento.

Gli eventi della classe "Actions" restituiscono un'istanza dello stesso tipo, permettendo quindi la concatenazioni di più invocazioni di metodi. Per eseguire la macro è presente il metodo "perform". E' anche possibile disaccoppiare la generazione dell'azione complessa dall'esecuzione tramite il metodo "build" di "Actions" che restituisce un'istanza di "Action", utile per eseguire più volte una stessa operazione.

Listato 3.35: Cambiamento del colore di sfondo dei collegamenti.

---

```

1 @Allora("^lo sfondo del collegamento cambia$") public void
  lo_sfondo_del_collegamento_cambia() {
2   WebElement bannerLink = page.getBannerLinks().get(0);
3   String bannerLinkColor = bannerLink.getCssValue("background-color");
4   assertNotEquals("rgba(0, 0, 0, 0)", bannerLinkColor);
5   assertEquals("rgba(91, 168, 42, 1)", bannerLinkColor);
6 }

```

---

Come si può intuire dalle asserzioni presenti nel passo, lo spostamento del cursore modifica il colore dello sfondo, verificando lo scenario.

### Selettori

Il metodo "findElement" restituisce il primo elemento, in ordine di apparizione all'interno del DOM, che soddisfa il selettore specificato tramite i metodi statici di "By". Per ottenere tutti gli oggetti "WebElement" validi è possibile utilizzare il metodo "findElements", come in listato 3.36.

Listato 3.36: Verifica della presenza di una dichiarazione alternativa per le immagini.

---

```

1 @Allora("^ogni collegamento ha una descrizione testuale$")
2 public void ogni_collegamento_ha_una_descrizione_testuale(){ List<
  WebElement> linkedImages = driver.findElements(By
    cssSelector("a img"));
3   for (WebElement linkedImage : linkedImages) {
4     /*...*/
5   }
6 }

```

---

Fra i selettori offerti da "By" è presente anche la possibilità di utilizzare espressioni XPath, come mostrato nel successivo frammento 3.37.

<sup>13</sup>Il costruttore dell'azione con un solo parametro di tipo "WebDriver" utilizza le azioni implementate da Selenium di default, che permette di simulare eventi tramite tastiera e mouse.

Listato 3.37: Selettore definito tramite espressione XPath.

---

```
1 WebElement findByXPath(String xpathExpression){    return driver.  
    findElement(By.xpath(xpathExpression));  
2 }
```

---

### 3.4.2 Debug con Selenium

Come per Capybara, l'esecuzione di test in modalità debug non offre molte informazioni utili per individuare eventuali errori nell'applicazione o nell'implementazione dei test. Avendo utilizzato nuovamente PhantomJS, che essendo un browser headless non possiede una GUI, è stato necessario combinare la possibilità di catturare le schermate e l'introduzione di breakpoint per valutare eventuali errori.

## 3.5 Definizione del modello

Listato 3.38: 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

```

---

La funzionalità corrente introduce in SBlog le operazioni CRUD per i post. Introdurre la gestione delle entità in un'applicazione web in Spring coincide con l'implementazione delle componenti descritte all'interno del capitolo introduttivo sul framework MVC.

### 3.5.1 Dipendenze

Come già riscontrato nell'implementazione della stessa funzionalità per RBlog, introdurre le entità e le relative operazioni richiede l'introduzione di meccanismi per garantire che i test di accettazione siano ripetibili ed indipendenti fra loro.

**Hooks** Con gli hook presenti in Cucumber è possibile definire delle callback da associare ad un certo punto dell'esecuzione dei test. Rispetto all'implementazione in Ruby e grazie alle funzionalità di Selenium non è stato necessario effettuare la cancellazione delle sessioni HTTP create tramite l'autenticazione. Sono stati infatti definiti due metodi, "setUpWebDriver" e "releaseWebDriver", descritti in listato 3.39, eseguiti rispettivamente prima e dopo ogni scenario, responsabili della creazione di una nuova finestra, per la quale è anche specificata una dimensione, e della successiva chiusura. Questa pratica, sicuramente costosa in termini di tempo d'esecuzione, permette però di eliminare le sessioni esistenti del browser ed evitare l'esecuzione di test in finestre precedentemente utilizzate per altri scenari.

Listato 3.39: Apertura e chiusura di una nuova "finestra" di PhantomJS.

---

```

1 @Before
2 public void setUpWebDriver() {    driver = new PhantomJSDriver();
   assertNotNull(driver);  driver.manage().window().
   setPosition(new Point(0, 0));  driver.manage().window().setSize
   (new Dimension(2048, 2048));
3 }
4 @After(order = 0)
5 public void releaseWebDriver() {
6     driver.close();
7 }

```

---

Per implementare un hook in Cucumber JVM è necessario dichiarare un metodo pubblico ed utilizzare una o più delle annotazioni fornite. Le annotazioni utilizzabili forniscono le stesse funzionalità descritte per la versione in Ruby e permettono di indicare sia per quali tag è necessario eseguire le callback, sia un attributo order che ne stabilisce la priorità d'esecuzione, nell'esempio il valore 0 specifica che il metodo "releaseWebDriver" verrà eseguito sempre come ultima operazione di uno scenario.

Listato 3.40: Rimozione dei post “Lorem Ipsum” creati nello scenario.

---

```

1  @After(value = "@clear")
2  public void clearIpsums() {
3      driver.navigate().to(getSblogURL());
4
5      String loremIpsumPostTitle = "Lorem Ipsum";
6      String xpathExpression =
7          String.format("//div[@class = 'post'][p/a[contains(text(), '%s')
8              ]]",
9              loremIpsumPostTitle);
10
11     List<WebElement> postDivs = driver.findElements(By.xpath(
12         xpathExpression));
13     for (WebElement postDiv : postDivs){
14         postDiv.findElement(By.className("remove_post_button")).click()
15     }
16 }

```

---

La callback “clearIpsums” in listato 3.40 invece, effettua la rimozione dei post con titolo contenente la stringa “Lorem Ipsum” in maniera simile a quanto effettuato per RBlog.

### 3.5.2 Gestione dei form

Selenium fornisce un'interfaccia lineare ed intuitiva per l'utilizzo dei form. Nel successivo metodo in listato 3.41, che descrive una generica operazione di inserimento del testo, si può osservare come non sia necessario compiere operazioni particolari per individuare nel DOM e utilizzare i campi di input di un “form” HTML.

Listato 3.41: Compilazione del titolo per un post.

---

```

1  protected void insertPostTitle(String title) {
2      WebElement titleInput = driver.findElement(By.id("post_title"));
3      titleInput.clear();
4      titleInput.sendKeys(title);
5  }

```

---

Il metodo “insertPostTitle” inserisce il testo passato come parametro all'interno del form per la creazione o modifica di un articolo<sup>14</sup>. Oltre alla compilazione testuale di un form, Selenium implementa le funzionalità per selezionare i checkbox, scegliere una o più opzioni attraverso i menu a tendina ed effettuare il caricamento di file.

### Verificare gli errori

A differenza di RSpec, la cui struttura prevede 4 moduli organizzati per funzionalità, JUnit ha una struttura monolitica.

Per definire un test con JUnit è necessario annotare con “@Test” un metodo pubblico, utilizzando però Cucumber non è possibile far corrispondere a ciascun

---

<sup>14</sup>Il metodo “clear” è necessario per gestire le operazioni di modifica, infatti il metodo “sendKeys” si limita ad aggiungere il testo al campo, concatenando le stringhe.

scenario un singolo test. Quali metodi corrispondano all'implementazione di uno scenario è verificato dinamicamente dal runner di Cucumber che non rispetta quindi la struttura classica prevista da JUnit.

Listato 3.42: Verifica di errori tramite JUnit.

```
1 @Test(expected=IndexOutOfBoundsException.class)
2 public void segFault() {
3     /*...*/
4 }
5
6 @Rule
7 public ExpectedException exception = ExpectedException.none();
8
9 @Test
10 public void segFault2() {
11     /*...*/
12     exception.expect(IndexOutOfBoundsException.class);
13     //Faulty expression
14 }
```

In assenza della corrispondenza “@Test” - scenario, non è possibile utilizzare le funzionalità di JUnit implementate tramite annotazione, come la verifica delle eccezioni o la definizione di “@Rule”~[69].

Listato 3.43: Un possibile “work-around”.

```
1 try {
2     visitHomePage();
3     findPostDivByTitle(postTitle);
4     fail();
5 } catch (NoSuchElementException e) {
6     //
7 }
```

Per definire asserzioni che verifichino la presenza di errori, nell'esempio in listato 3.43 è verificato che nella home-page di SBlog non sia presente un post con titolo uguale al parametro, è possibile utilizzare una combinazione del metodo statico “fail” della classe “Assert” di JUnit e un blocco “try-catch” sull'errore desiderato.

## 3.6 Login & Autorizzazione

Listato 3.44: Descrizione della funzionalità di autenticazione.

---

```

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

```

---

Per introdurre la gestione dell'autenticazione in SBlog è stato introdotto un controllo ed una vista per effettuare il login. Come per RBlog, le azioni prevedono la verifica dell'autorizzazione; ad esempio non è possibile effettuare il logout se l'utente non è autenticato.

L'entità "Session", per cui è stato definito il controllo, non è presente nel modello ma mantiene esclusivamente lo stato della navigazione, mantenendo all'interno della sessione HTTP un riferimento all'identificatore dell'utente.

Listato 3.45: Azione per la gestione del logout.

---

```

1 @RequestMapping(value = "/logout", method = RequestMethod.GET)
   public String destroy(Model model, RedirectAttributes
   redirectAttributes, HttpSession httpSession){
2   if(sessionService.isLogged(httpSession)){
3     redirectAttributes.addFlashAttribute("content_template",
       "/posts/index");
4     redirectAttributes.addFlashAttribute("notice", "Arrivederci!");
5     httpSession.invalidate();
6   }
7   return "redirect:/";
8 }

```

---

In listato 3.45, il metodo "destroy", dichiarato all'interno della classe Session-Controller, verifica il valore della sessione ed eventualmente invalida il contenuto. L'azione non restituisce un riferimento ad una vista, ma reindirizza la navigazione verso l'home-page di SBlog, tramite la concatenazione della stringa "redirect:" e l'indirizzo relativo di destinazione. Al contrario di RoR, che definisce i "Filter", non è previsto una funzionalità per applicare delle callback per le azioni dei controlli.

Per poter sfruttare le funzionalità di Spring per la gestione della sessione, è necessario aggiungere un parametro "HttpSession" alla segnatura del metodo, che sarà opportunamente istanziato dal framework con i valori correnti della navigazione.

Listato 3.46: Login tramite Selenium.

---

```

1 @Dato("^mi autentico come \".*?\".$")
2 public void mi_autentico_come(String email) {
3   WebElement loginLink = findById("log_in_link");
4   loginLink.click();
5   WebElement emailInputElement = driver.findElement(By.name("email"
6   ));
7   WebElement passwordInputElement = driver.findElement(By.name("
   password"));

```

---



```
7     emailInputElement.sendKeys(email);  
8     passwordInputElement.sendKeys("password");  
9     emailInputElement.submit();  
10 }
```

---

Per effettuare il login in SBlog, come in listato 3.46, è sufficiente inserire email e password nell'apposito form, operazioni effettuate tramite il metodo "sendKeys" già mostrato in precedenza. Il metodo, invocabile tramite un numero variabile di "CharSequence", simula pressione dei tasti. Non è stato verificato nei test, ma la documentazione del metodo indica che gli eventi relativi alla pressione del tasto, "keyup", "keydown" e "keypress", sono scatenati per ogni carattere della stringa.

Una funzionalità interessante di Selenium è la possibilità di inviare il contenuto del form, invocando il metodo "submit" su uno qualunque degli elementi inclusi in esso, come mostrato in listato 3.46.

All'interno della libreria sono presenti due classi per la gestione dei cookie~[90]: "Cookie" e "Cookie.Builder". Le funzionalità presenti permettono la creazione, cancellazione, ricerca e modifica dei cookie esistenti. Come per RBlog però, le sessioni in Spring non espongono i valori in chiaro e non sono state effettuate operazioni di codifica e decodifica, evidentemente in contrasto con lo stile dei black box test.

## 3.7 Asincronia

In questa sezione sono descritti i test di accettazione effettuati per verificare le funzionalità di Selenium nella gestione di comportamenti asincroni.

### 3.7.1 JavaScript

L'obiettivo della prima funzionalità è verificare il comportamento di Selenium con un comportamento asincrono ma estremamente rapido. Come per RBlog, è aggiunto, tramite una chiamata ad una funzione JavaScript associata, un piccolo logo al piè di pagina.

Listato 3.47: Introduzione di un breve script Javascript.

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

Per ricercare all'interno del DOM un elemento che viene aggiunto dinamicamente è possibile implementare un meccanismo implicito di attesa, in maniera simile a Capybara, impostando un tempo massimo di attesa per ogni ricerca. In tal caso, però, non è però possibile impostare una frequenza di polling. Alternativamente utilizzare delle attese esplicite come in listato 3.48.

Per l'implementazione dei test è stato scelto di utilizzare quest'ultima soluzione. Per istanziare un oggetto di tipo "WebDriverWait"~[91] è necessario impostare un tempo di attesa massimo. Tramite il metodo "until" si interrompe l'esecuzione dello scenario quando è verificata la condizione espressa dal parametro.

Listato 3.48: Verifica della presenza del logo nel piè di pagina.

```
1 @Allora("^è presente il logo$")
2 public void è_presente_il_logo() {
3     new WebDriverWait(driver, 2).until(ExpectedConditions
4         presenceOfElementLocated(By.id("woodstock")));
5     page.footer.findElement(By.cssSelector("img"));
6 }
```

Nel listato 3.48 è descritta una condizione, corrispondente al metodo statico "presenceOfElementLocated" della classe "ExpectedConditions", che effettua la verifica della presenza di un certo identificatore all'interno del DOM<sup>15</sup>.

La classe "ExpectedConditions" include diversi metodi per esprimere le attese esplicite in funzione di diverse condizioni, dalle più semplici che verificano la presenza di un elemento o di un testo nel DOM, fino all'attesa per una finestra di dialogo o elemento cliccabile.

<sup>15</sup>Non è possibile attualmente definire una condizione che verifichi l'esistenza di un selettore all'interno di un particolare elemento del DOM.

### 3.7.2 Scenari sull'auto-completamento con JQuery UI

Rispetto alla funzione Javascript della precedente funzionalità, l'utilizzo del widget di JQuery UI per l'auto-completamento del menù introduce un overhead significativo nel caricamento della pagina, suddiviso fra tempo di completamento della richiesta HTTP tramite AJAX, l'esecuzione della funzione in JQuery e la visualizzazione del menù nel browser web.

Listato 3.49: Scenario riguardante l'auto-completamento della ricerca.

---

```

1 Scenario: Autocompletamento della ricerca
2   Dato nell'intestazione è presente la barra di ricerca
3   Dato il post "Lorem Ipsum" esiste
4   Quando inserisco il testo "lor" da ricercare
5   Allora viene proposto il post "Lorem Ipsum"
6   Quando inserisco il testo "xyz" da ricercare
7   Allora non è proposto alcun post
8   ...

```

---

Nel successivo listato 3.50 è utilizzato un metodo per l'attesa esplicita per individuare i suggerimenti visualizzati per il menu.

Listato 3.50: Verifica dei titoli suggeriti.

---

```

1 @Allora("^viene proposto il post \"(.*)\"") public void
   viene_proposto_il_post(String postTitle) {
2   String xpathExpression = String.format("//li[@class = 'ui-menu-
   item']", postTitle);
3   List<WebElement> lis = new WebDriverWait(driver, 4).until(
       ExpectedConditions.presenceOfAllElementsLocatedBy(By.xpath(
           xpathExpression)));
4   boolean isPostProposed = false;
5   for (WebElement li : lis) {       isPostProposed |= li.getText().
       equals(postTitle) && li.isDisplayed();
6   }
7   assertTrue(isPostProposed);
8 }

```

---

Il metodo "presenceOfAllElementsLocatedBy" restituisce una collezione di elementi identificati dal selettore, descritto tramite un'espressione XPath, ed attende entro il tempo massimo fissato che almeno un elemento sia presente nella pagina.

Ottenuti gli elementi, durante le iterazioni del ciclo for, è verificata la corrispondenza fra il testo del suggerimento e il titolo passato come parametro e l'effettiva visualizzazione dell'elemento nel browser.

L'implementazione del menu di auto-completamento di JQuery prevede che la collezione degli elementi che costituiscono i suggerimenti sia definita a priori. Al contrario, se il widget prevedesse la popolazione dinamica del menu scorrendo nella lista, il metodo "presenceOfAllElementsLocatedBy" non risulterebbe attendibile per verificare la presenza di nuove entry, perché la condizione risulterebbe già vera alla comparsa del primo elemento.

## Capitolo 4

# MVC 5

### 4.1 ASP.NET MVC 5

In questo capitolo è trattato lo sviluppo di CSBlog, lo sviluppo tramite il framework ASP.NET MVC 5~[92] del nostro caso di studio. L'implementazione dei test di accettazione già mostrata nei precedenti capitoli utilizzando Specflow~[93], una libreria per il BDD ispirata a Cucumber, e Coypu~[94] per automatizzare la navigazione del browser web.

In questo capitolo saranno descritte le particolarità del framework, l'implementazione delle diverse componenti e gli strumenti utilizzati. Dalla sezione 4.3 in poi sono documentate le diverse funzionalità presenti nella libreria dei test di accettazione e viene descritto il processo di testing, includendo esempi e frammenti di codice.

#### 4.1.1 Visual Studio 2013

Per lo sviluppo di CSBlog, dei test di accettazione e la gestione del modello è stato utilizzato Visual Studio 2013 Ultimate, nel seguito VS, che integra le diverse funzionalità per lo sviluppo web, la configurazione della propria applicazione e l'implementazione delle diverse componenti del pattern.

Tutti i plugin necessari, come ad esempio quello per Specflow, sono stati installati tramite NuGet~[96], il gestore di pacchetti per il mondo .NET, direttamente tramite l'ambiente di sviluppo. VS è un ambiente di sviluppo con enormi potenzialità, articolato e complesso, con numerose funzionalità per il supporto alla configurazione e generazione del progetto, come sarà mostrato nelle prossime sezioni.

## 4.2 L'interpretazione di ASP.NET del pattern MVC

In questa sezione è descritto il processo di definizione dell'architettura di CSBlog, caratterizzato da un uso costante degli strumenti di VS per la definizione delle varie componenti. Al termine della sezione risulterà evidente come, nonostante il framework MVC5 e l'intero progetto ASP.NET sia estremamente articolato, la presentazione delle grandi potenzialità messe a disposizione è sufficientemente accessibile così che lo sviluppo non ne risente.

### 4.2.1 Il modello

Per implementare l'interfaccia del modello di MVC5 è stato scelto l'Entity Framework, attualmente alla versione 6, che implementa l'astrazione del modello attraverso la tecnica ORM in maniera simile a quanto già visto per RoR e Spring. Il framework si integra con ADO.NET, responsabile dell'interazione con i sistemi di persistenza esistenti e del mantenimento della consistenza fra le entry dei database e gli oggetti.

L'EF~[97] supporta sia l'approccio "code-first", in cui si definiscono le classi che rappresentano le entità e da esse si genera il database corrispondente, in maniera analoga a quanto già visto per RoR e JPA, sia l'approccio "db-first", in cui si generano le classi a partire dal DB.

Per CSBlog è stato scelto quest'ultimo approccio, utilizzando il DDL di SQL -come database è stato scelto SQL Server Express, la versione gratuita del database sviluppato da Microsoft-, la generazione guidata e gli strumenti grafici per la gestione disponibili per creare le tabelle "Post" ed "Autore" e la relazione presente fra le due entità in breve tempo.

Visual Studio include un wizard per la configurazione del modello, che provvede a definire la connessione al database, a generare le classi parziali<sup>1</sup> rappresentanti le entità e a configurare le risorse necessarie per il funzionamento dell'EF.

Le classi che descrivono le entità dell'EF sono gestite attraverso un'estensione della classe "DbContext"~[98], che fornisce le funzionalità per eseguire interrogazioni, tener traccia dei cambiamenti allo stato delle istanze ed invocare le operazioni di creazione, aggiornamento e cancellazione.

Listato 4.1: Il contesto di CSBlog generato da VS.

---

```

1 public partial class CSBlogEntities : DbContext{
2     public CSBlogEntities() : base("name=CSBlogEntities"){
3         protected override void OnModelCreating(
4             DbModelBuilder modelBuilder){
5             throw new UnintentionalCodeFirstException();
6         }
7     }
8     public virtual DbSet<Author> Authors { get; set; }
9     public virtual DbSet<Post> Posts { get; set; }
10 }

```

---

<sup>1</sup>Una classe parziale in C# rappresenta una parte di una classe intera. La definizione completa è suddivisa in più file che saranno unite a tempo di compilazione.

In listato 4.1, le proprietà “Authors” e “Posts” rappresentano le entità presenti nel database per i rispettivi tipi, rappresentando quindi a livello OO le corrispondenti tabelle e sono utilizzate dall'EF per effettuare le query.

Per implementare le interrogazioni necessarie per CSBlog è stato utilizzato LINQ, Language-Integrated Query.

Listato 4.2: Proiezione dei titoli dei post con LINQ.

```
1 CSBlogEntities db = new CSBlogEntities();  
2 IQueryable<Post> qTitles = db.Posts.Select(p => p.title)  
3 var postTitles = qTitles.ToList();
```

LINQ~[99] permette di definire le proprie query attraverso un insieme di metodi che rispecchiano le funzionalità di SQL e permettono di utilizzare il paradigma ad oggetti, ad esempio la proiezione per ottenere i titoli dei post esistenti è definita effettuato un accesso all'attributo “title” dell'entità “Post”. Utilizzando una libreria che fornisce una tale astrazione del modello e delle entità è possibile definire le proprie interrogazioni sfruttando tutte le funzionalità esistenti per C# in VS, come l'auto-completamento e l'evidenziatura dei diversi elementi sintattici. Inoltre LINQ definisce query utilizzando metodi generici che permettono di verificare a compile-time il corretto utilizzo dei tipi e degli attributi presenti nelle interrogazioni.

Una query LINQ non viene eseguita al momento della sua definizione, ma solo quando è necessario accedere ai suoi elementi, ad esempio per iterarvi sopra, accedere al primo elemento (metodo “First”) o verificare che ne esista almeno uno (metodo “Any”).

In maniera simile all'interfaccia per la definizione delle query con gli Active Record di RoR, lo sviluppatore può sfruttare l'astrazione introdotta dall'uso dell'EF e definire le proprie query in maniera veloce ed intuitiva. Rispetto però alla controparte in Ruby, LINQ è più completo permettendo la definizione di interrogazioni anche su altri sistemi per la persistenza, come documenti XML, ed anche su collezioni di tipo “IEnumerable” in memoria.

La semplice query “qTitles”, in listato 4.2, di proiezione dell'esempio è suddivisibile in tre elementi: l'ottenimento del contesto tramite le classi dell'EF, rappresentato dall'oggetto “db” e corrispondente alla sorgente di dati o del DB, la selezione della proprietà Posts, per specificare quale sia l'entità utilizzata nell'interrogazione, la specifica delle operazioni, come il metodo “Select”.

L'esecuzione della query coincide con l'invocazione del metodo “ToList”. L'oggetto “qTitles” ha tipo “IQueryable”~[100], interfaccia generica sul tipo dell'interrogazione che estende “IEnumerable”~[101] e dichiara le funzionalità per compiere la valutazione della query.

Oltre alla generazione dell'interfaccia del modello, VS include un browser per visualizzare lo schema delle entità esistenti e le relazioni presenti. Tramite questo strumento, direttamente nell'ambiente di sviluppo, è possibile visualizzare e aggior-

nare la struttura del modello, sincronizzare gli schemi dei database utilizzati, ed anche gestire associazioni e relazioni di ereditarietà.

### 4.2.2 I controlli

Come per la definizione del modello, anche per i controlli e le relative azioni, sono stati utilizzati gli strumenti di VS per la generazione automatica delle componenti. Il wizard per la configurazione è ben strutturato e permette di indicare quali azioni sia necessario generare, ad esempio è possibile creare un controllo contenente le operazioni CRUD oppure vuoto, e se utilizzare l'EF per associare il nuovo controllo ad una delle entità e generare delle azioni opportunamente connesse al modello.

Durante il processo di configurazione del nuovo controllo è anche generato un'insieme di viste corrispondenti alle azioni create. In maniera simile a RubyMine, gli strumenti dell'ambiente di sviluppo facilitano lo sviluppo fornendo un'implementazione funzionante per le nuove applicazioni e preziose indicazioni sulla struttura del framework e di come le componenti interagiscano fra loro.

Per semplificare la risoluzione delle richieste HTTP per l'applicazione, MVC5 definisce un file "RouteConfig.cs" all'interno della cartella "AppData".

Listato 4.3: Definizione del pattern per la risoluzione delle richieste.

---

```

1 public class RouteConfig{
2     public static void RegisterRoutes(RouteCollection routes){
3         /*...*/
4         routes.MapRoute(
5             name: "Default",
6             url: "{controller}/{action}/{id}",
7             defaults: new {
8                 controller = "Post",
9                 action = "Index",
10                id = UrlParameter.Optional
11            }
12        );
13    }
14 }
```

---

Il metodo "RegisterRoutes" si occupa di registrare la convenzione scelta per la risoluzione delle richieste effettuate dagli utenti. Come si può osservare dal metodo, il pattern scelto rispecchia l'organizzazione di un'architettura REST: ad esempio l'URL relativo "/Post/Details/id", per visualizzare il contenuto di un singolo post, è suddiviso in tre elementi, il nome del controllo, l'azione corrispondente e l'identificatore dell'elemento. Inoltre è specificato quale sia l'homepage tramite il parametro "default".

Listato 4.4: Visualizzazione di un singolo post.

---

```

1 namespace Blog.Controllers {
2     public class PostController : Controller    {
3         private CSBlogEntities db = new CSBlogEntities();
4         /*...*/
5     }
```

---

```

5     public ActionResult Details(Guid? id){
6         if (id == null){
7             return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
8         }
9         Post post = db.Posts.Find(id);
10        if (post == null){
11            return HttpNotFound();
12        }
13        return View(post);
14    }
15    /*...*/
16 }
17 }

```

Per implementare un nuovo controllo, è sufficiente creare una nuova classe pubblica all'interno del namespace del progetto ed estendere la classe astratta "Controller"~[103]. Nel frammento della classe "PostController" in listato 4.4 è mostrata l'implementazione di una singola azione che, in funzione del parametro "id" carica tramite LINQ ed EF l'entità corrispondente e restituisce un riferimento alla vista "Details".

Infatti il tipo "ActionResult" utilizzato come risultato dell'azione ha il compito di rappresentare il risultato e permettere al framework di fornire una risposta; nel frammento il metodo "View" della classe padre "Controller" istanzia un'oggetto "ViewResult" che specifica la vista da renderizzare.

Ad ogni azione, se non specificato diversamente, corrisponde una vista con ugual nome. Nel nostro esempio, dopo aver individuato il corretto "Post" all'interno del modello, il server web fornirà all'utente la vista corrispondente al file "Views/Details.cshtml". L'azione "Details" accetta richieste HTTP di qualsiasi tipo, per applicare delle limitazioni è sufficiente annotare il metodo con attributi come "HttpGet" o "HttpPost" presenti in MVC5.

Rispetto a Spring e RoR, rispettivamente con i Service e gli Helper, non è prevista all'interno dell'architettura una componente con il ruolo di disaccoppiare i controlli dalla logica per l'accesso al modello, sta quindi all'utente introdurre il pattern che meglio si presta allo scopo ed implementarlo.

### 4.2.3 Le viste

Per l'implementazione delle viste di CSBlog è stato utilizzato Razor~[102], un linguaggio ASP.NET per l'implementazione di pagine dinamiche che introduce un DSL basato su C# ed integrato in VS, ad esempio è previsto l'IntelliSense e la possibilità di creare nuove pagine sfruttando i wizard presenti nell'ambiente di sviluppo.

Razor rappresenta la scelta di default per lo sviluppo di una nuova applicazione web, introdotto dalla versione 4 del framework, introduce una sintassi per ottimizzare la generazione di pagine HTML, definendo un insieme di costrutti per permettere allo sviluppatore di distinguere facilmente le porzioni di codice dinamico dal resto del documento statico.

---

Listato 4.5: Hello Razor!



```

1 @{
2     var helloRazor = "Benvenuto su CSBlog!";
3     var weekDay = DateTime.Now.DayOfWeek;
4     var greetingMessage = greeting + " Oggi è: " + weekDay;
5 }
6 <p>@helloRazor</p>

```

Il design delle istruzioni è essenziale ed è paragonabile ad ERB, sono infatti presenti anche in Razor dei delimitatori per includere all'interno delle viste codice. Nel frammento d'esempio è introdotto un blocco contenente istruzioni C#, all'interno del quale è creato un messaggio di benvenuto. Tramite la notazione "@variabile" è possibile ottenere il valore della stringa definita nel blocco.

La distinzione fra la sintassi "@{...}" e l'operatore "@", che consente di accedere ai valori e di utilizzarne le espressioni per generare le viste dinamiche, permette di dichiarare viste in cui sono distinte in maniera chiara le porzioni di codice che includono l'implementazione della logica delle istruzioni che esclusivamente utilizzano le variabili in lettura.

Listato 4.6: Il ciclo "for" in Razor.

```

1 <ul>
2 @for (int i = 0; i < 10; i++){
3     <li>@i</li>
4 }
5 </ul>

```

In listato 4.6, la sintassi di Razor include anche i più comuni costrutti sintattici, è mostrato l'utilizzo di un ciclo "for" le cui iterazioni generano nuovi elementi della lista "ul".

L'organizzazione e le convenzioni utilizzate da MVC5 semplificano la definizione dell'intera applicazione, ed anche per la definizione delle viste è previsto un meccanismo per fattorizzare al meglio il proprio codice.

Per ogni vista da processare e visualizzare in risposta ad una richiesta HTTP, viene ricercato il file "\_ViewStart.cshtml"<sup>2</sup> all'interno della cartella "Views". In CSBlog è stata utilizzata questa risorsa per indicare quale sia il file Razor contenente il layout delle pagine.

Listato 4.7: Il contenuto della vista \_ViewStart.

```

1 @{
2     Layout = "~/Views/Shared/_Layout.cshtml";
3 }

```

La scelta del layout può essere effettuata dinamicamente come in listato 4.7, permettendo di aver maggior controllo sull'aspetto delle proprie pagine semplicemente cambiando il percorso della risorsa o aggiungendo della logica all'assegnazione. Ad esempio è possibile definire più versioni del proprio sito, ottimizzandone l'uso in funzione del tipo di device che sta effettuando la navigazione.

<sup>2</sup>"cshtml" è l'estensione delle viste in Razor.

Listato 4.8: La struttura delle viste in CSBlog.

```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3   @Html.Partial("_HeadPartial")
4   <body>
5     @Html.Partial("_HeaderPartial")
6     <div id="content">
7       @RenderBody()
8     </div>
9     @Html.Partial("_FooterPartial")
10  </body>
11 </html>
```

Per definire un layout è sufficiente creare una nuova vista con Razor ed utilizzare funzionalità come “Partial”, per espandere il contenuto di una vista parziale, o “RenderBody” per espandere la vista restituita dall'azione che ha gestito la richiesta. Nell'esempio è mostrato com'è organizzata una pagina di CSBlog.

#### 4.2.4 Peculiarità

Visual Studio, ASP.NET e l'EF sono software estremamente ricchi di funzionalità e professionali, ma il loro utilizzo è semplificato da VS che, alla creazione di una nuova applicazione, definisce un progetto nel quale sono utilizzate la maggior parte delle componenti presenti nel framework. Navigando fra i sorgenti appena creati è possibile approfondirne le funzionalità, intuirne le potenzialità e osservare le soluzioni proposte dal team di VS.

Oltre ad includere un esempio di applicazione funzionante per ogni nuovo progetto, VS fornisce agli sviluppatori numerosi wizard per generare e configurare le nuove funzionalità. Durante lo sviluppo non è mai stato necessario modificare manualmente i file XML per la configurazione dell'applicazione e l'introduzione di nuovi controlli, viste o entità non ha richiesto uno studio approfondito del framework per una singola funzionalità, come ad esempio è successo per la definizione del modello con JPA.

Inoltre VS integra all'interno dell'ambiente di sviluppo tutte le diverse categorie di strumenti e linguaggi necessari per lo sviluppo di un'applicazione web. Oltre ai classici HTML, CSS e Javascript, è supportato di default lo sviluppo con tecnologie innovative come Sass e Bootstrap~[104]<sup>3</sup>.

---

<sup>3</sup>Bootstrap è un'interessante framework, sviluppato originariamente da Twitter, per la definizione di pagine web responsive. E' particolarmente apprezzato per la definizione di siti facilmente navigabili da device mobile in quanto non prevede l'utilizzo di fogli di stile per definire la disposizione degli elementi nella pagina.

## 4.3 Hello CSBlog!

In questa sezione sono introdotti gli strumenti per effettuare la definizione dei test di accettazione automatici per CSBlog. Nella sezione 4.3.1 è descritto il processo di configurazione di SpecFlow, il framework per il BDD, e nella sezione 4.3.2 quello di Coypu~[94], la libreria per l'automazione della navigazione web scelta per .NET. Successivamente sono introdotti i primi test e le funzionalità utilizzate per l'implementazione.

### 4.3.1 SpecFlow

SpecFlow è un framework per .NET che sfrutta alcune componenti di Cucumber come il DSL e il relativo parser, entrambe presenti pubblicamente su GitHub. Nonostante utilizzi Gherkin, non è un porting dell'intero framework in C#, al momento disponibile solo nelle versioni Java, Ruby e JavaScript.

#### Le funzionalità

Listato 4.9: La prima feature di SBlog

---

```

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

```

---

Grazie all'utilizzo di SpecFlow è possibile riutilizzare gli stessi acceptance test utilizzati per lo sviluppo di RBlog e SBlog. Nell'importazione e nello sviluppo dei test non sono state rilevate problemi legati al linguaggio ed ai costrutti sintattici utilizzati, sono però presenti delle differenze nell'implementazione, come verrà descritto nelle successive sezioni.

#### Supporto a Specflow in VS

Per utilizzare SpecFlow in VS è sufficiente installare tramite il gestore dei pacchetti il relativo plugin. Per eseguire le funzionalità e gli scenari tramite il runner di NUnit~[95] è necessario anche un pacchetto aggiuntivo, installabile tramite la console di NuGet utilizzando il comando seguente.

Listato 4.10: Installazione del plugin per SpecFlow in VS.

---

```

1 Install -Package SpecFlow.NUnit

```

---

Completata l'installazione del software non è necessario compiere alcuna operazione di configurazione. Il plugin estende il funzionamento di VS aggiungendo il completo supporto a Gherkin, evidenziando la sintassi nelle diverse lingue esistenti, introducendo l'auto-completamento all'interno dei file ".feature", permettendo la navigazione da passo ad implementazione, permettendo la generazione automatica degli stub rappresentanti i passi e aggiungendo le funzionalità in Gherkin nei menù

rapidi per la creazione delle risorse.

Per l'esecuzione delle funzionalità è possibile selezionare direttamente i file ".feature", eventualmente utilizzando il tag "@ignore" per saltare alcuni scenari, oppure sfruttare l'integrazione con NUnit per scegliere quale sotto-insieme di scenari eseguire. E' possibile utilizzare i diversi ordinamenti di NUnit per eseguire gli scenari in funzione di durata di esecuzione, risultato del test, tag utilizzati o namespace.

### 4.3.2 Coypu

I criteri utilizzati per scegliere la libreria per l'automazione della navigazione via browser in questa tesi, favoriscono strumenti scritti nel linguaggio utilizzato per implementare l'applicazione web, il cui sviluppo sia attivo, open-source e che abbiano una comunità attiva per poter avere un riscontro in caso di difficoltà.

L'individuazione di una libreria per .NET che soddisfacesse questi parametri non è stato semplice. Oltre a Selenium, già utilizzato per Java, sono stati valutati Watin, il cui sviluppo è fermo dal 2011, e Telerick Testing Framework, che per quanto sia un progetto attivo e ben documentato, sembra non essere utilizzato ed essere privo di una propria comunità di utenti.

Viste le diverse difficoltà nella scelta di una libreria è stata effettuata una scelta sperimentale utilizzando Coypu, un wrapper di Selenium Web Driver, scritto in C# che nella propria implementazione si ispira al DSL di Capybara.

Il progetto è stato rilasciato la prima volta 2011, 2 anni prima della versione beta di Selenium 2.0, il cui sviluppo è ancora attivo, anche se il supporto è principalmente effettuato dal un singolo autore. Nonostante si tratti di una libreria open-source utilizzata da una ridotta comunità, presenta un'interessante prospettiva dell'automazione della navigazione web che merita di essere approfondita.

I principali obiettivi di Coypu riguardano la semplificazione e razionalizzazione delle funzionalità della libreria di Selenium e la definizione di una libreria che, come Capybara, permetta di descrivere i propri test nella maniera più vicina possibile a come un utente descriverebbe le proprie azioni in termini di interazioni con il browser.

#### Configurazione di una sessione di testing

Come effettuato per Selenium in SBlog, anche per Coypu è stato scelto di utilizzare un nuovo ambiente di testing per ogni scenario, aprendo una nuova finestra prima dell'esecuzione e rilasciando le risorse al termine.

Listato 4.11: Creazione della sessione.

```
1 [BeforeScenario]
2 public void Before(){
3     var sessionConfiguration = new SessionConfiguration
4     {
5         Port = 1448,
6         Driver = typeof(SeleniumWebDriver),
```

```

6      Browser = Coypu.Drivers.Browser.PhantomJS,
7      Timeout = TimeSpan.FromSeconds(5),
8      RetryInterval = TimeSpan.FromSeconds(0.1)
9  };
10
11  _browser = new BrowserSession(sessionConfiguration);
12      _browser.MaximiseWindow();
13      _objectContainer.RegisterInstanceAs(_browser);
14 }

```

Anche in SpecFlow è possibile definire delle callback, nell'esempio 4.11 è definito un hook "BeforeScenario" da eseguire prima di ogni scenario, contenente le istruzioni per la definizione di una sessione. In particolare tramite la classe "Session-Configurazione" è impostata la porta del server web, il driver ed il browser utilizzato ed il tempo massimo di esecuzione di ricerca per un selettore all'interno della pagina.

Listato 4.12: Chiusura del browser.

```

1  [AfterScenario]
2  public void AfterScenario(){
3      /*...*/
4      _browser.Dispose();
5  }

```

Per rilasciare la sessione di testing è invocato il metodo "dispose" al termine di ogni scenario, come si vede in listato 4.12.

### PhantomJS

Per utilizzare PhantomJS come browser per i propri test, è necessario mantenere l'eseguibile all'interno del "path" del proprio sistema o all'interno della cartella "bin" del progetto in VS. Per installare il browser localmente è possibile utilizzare la console di Nuget ed eseguire il seguente comando.

Listato 4.13: Installazione locale di PhantomJS.

```

1  install -package phantomjs.exe

```

### 4.3.3 Implementazione dei passi

Listato 4.14: Implementazione del passo "apro SBlog".

```

1  [Given(@"apro CSBlog")]
2  public void DatoAproCSBlog(){
3      /*...*/
4  }

```

In maniera paragonabile a Cucumber e Cucumber-JVM, l'implementazione di un passo coincide con la definizione di un metodo pubblico annotato con un attributo, come Given in listato 4.14, per indicare tramite l'espressione regolare quale sia il passo implementato. I passi devono essere inseriti all'interno di classi annotate con l'attributo "Binding".

Rispetto ai framework per il BDD utilizzati finora, Specflow è più rigido per quanto riguarda l'implementazione dei passi. Infatti obbligatorio che il tipo dichiarato in Gherkin coincida con l'attributo utilizzato. Quindi se la stessa espressione viene utilizzata in due tipi di passi, si deve annotare il metodo che li implementa con entrambe gli attributi, come in listato 4.15.

Listato 4.15: Implementazione valida per una pre-condizione ed un evento.

---

```

1 [Binding]
2 public class Constraints : BaseStep      {
3     [Given(@"mi autentico come "(.*)""")]
4     [When(@"mi autentico come "(.*)""")]
5     public void DatoMiAutenticoCome(string email){
6         /*...*/
7     }
8     /*...*/

```

---

Inoltre è possibile definire lo scope per ogni metodo o classe che implementi le funzionalità scritte in Gherkin tramite l'attributo "Scope". E' possibile impostare la visibilità dei passi in funzione del diverso posizionamento dell'attributo, annotando l'intera classe o il singolo metodo. Il codice in listato 4.16 vincola la visibilità esprimendo le diverse condizioni disponibili.

Listato 4.16: Definizione dell'attributo Scope.

---

```

1 [Scope(Tag = "xyz", Feature = "Funzionalità", Scenario = "Prova")]

```

---

## Navigare all'interno del sito

Listato 4.17: Apertura di CSBlog.

---

```

1 [Given(@"apro CSBlog")]
2 public void DatoAproCSBlog(){
3     browser.Visit("/");
4     Assert.AreEqual("CSBlog", browser.Title);
5 }

```

---

Come in Selenium, Coypu fornisce all'utente diverse funzionalità per navigare all'interno del browser. Tramite l'istanza di "BrowserSession", che include le funzionalità per la gestione del browser come il ridimensionamento della finestra, il refresh della pagina e la navigazione attraverso la cronologia, è possibile navigare esplicitamente ad un URL passato come parametro, nell'esempio in listato 4.17 è mostrato come sia anche possibile utilizzare un percorso relativo.

Listato 4.18: Alcune delle funzionalità presenti nella classe "BrowserWindow".

---

```

1 public void AcceptModalDialog(Options options = null);
2 public void CancelModalDialog(Options options = null);
3 public bool HasDialog(string withText, Options options = null);
4 public bool HasNoDialog(string withText, Options options = null);

```

---

Approfondendo le funzionalità incluse nella classe "BrowserWindow", come ad esempio quello in listato 4.18, si intuisce la propensione di Coypu per la definizione di

metodi che rappresentino le normali interazioni di un'utente con il browser durante la navigazione web e siano il più intuitivi possibili.

Listato 4.19: Navigazione tramite un collegamento in Coypu.

```
1 [When(@"navigo verso "(.*)""")]  
2 public void QuandoNavigoVerso(string pageName){  
3     Assert.True(browser.FindLink(pageName).Exists());  
4     browser.ClickLink(pageName);  
5 }
```

Inoltre Coypu organizza la libreria in maniera che le funzionalità utilizzate di frequente siano facilmente accessibili. Ad esempio per utilizzare un collegamento presente nella pagina per navigare all'interno del sito è sufficiente utilizzare il metodo "ClickLink" esistente all'interno della classe "BrowserWindow" che rappresenta la sessione di navigazione ed è quindi già istanziata al momento dell'esecuzione dei test, come illustrato in listato 4.19.

Inoltre il metodo "ClickLink" effettua il click dell'elemento in funzione del testo che appare e non di un selettore CSS o XPath, facilitando la definizione di test che mantengono la stessa prospettiva di un utente.

Il metodo "FindLink" permette la ricerca all'interno di un DOM di un collegamento in funzione del testo fornito come parametro. Il metodo restituisce un'istanza di "ElementScope" che rappresenta un tag presente nel DOM. In maniera simile a Capybara, sono presenti numerose proprietà per ottenere informazioni sull'elemento ed i metodi per eseguire le azioni.

Al fine di semplificare la definizione dei test, Coypu include in tutte le funzionalità che analizzano il DOM un meccanismo di attesa implicito. Assumere che ogni elemento possa essere il risultato di un'operazione asincrona, permette all'utente di definire test più semplici e mantenibili. Come definito nel metodo 4.11, che rappresenta la creazione e configurazione di una nuova sessione per la verifica di uno scenario, è possibile impostare sia il valore di attesa massimo che la frequenza di polling desiderata.

Una carenza riscontrata nelle prime fasi di sviluppo è l'assenza di documentazione soddisfacente. Nonostante in Coypu sia facile intuire il funzionamento della libreria leggendo la segnatura dei metodi, la definizione di qualche esempio ne semplificherebbe l'utilizzo.

**Definire asserzioni con NUnit** NUnit è una libreria per la definizione di unit-test in C# nata dal porting di JUnit, attualmente alla versione 2.6 e il cui sviluppo è svolto in maniera separata dalla versione Java.

Nonostante i due progetti non siano più sviluppati dallo stesso team, utilizzando le asserzioni di NUnit è facile riconoscere l'origine comune. Ad esempio nel successivo passo in listato 4.20, sono utilizzati i metodi "AssertThat" e "AreEqual", che rispettivamente verificano la presenza di un elemento con identificatore HTML "logo" ed effettuano un confronto sul titolo della pagina.

Listato 4.20: NUnit AreEqual

---

```
1 [Then(@"posso tornare alla pagina iniziale")]
2 public void AlloraPossoTornareAllaPaginaIniziale(){
3     var logoLink = browser.FindId("logo");
4     Assert.That(logoLink.Exists());
5     logoLink.Click();
6     Assert.AreEqual("CSBlog", browser.Title);
7 }
```

---

Uno degli obbiettivi degli autori di NUnit è di definire dei metodi che, attraverso la propria segnatura, descrivano il proprio funzionamento e siano facilmente leggibili.

Listato 4.21: Un possibile uso dell'asserzione "That".

---

```
1 Assert.That(myString, Is.EqualTo("Hello"));
```

---

Il metodo "That" nell'esempio in listato 4.21, è utilizzato per verificare un valore booleano, in generale però è utilizzato per la sua caratteristica di accettare parametri "IConstraint". Questa interfaccia definisce un insieme di metodi per rappresentare una condizione e, oltre alle classi di vincoli fornite dalla libreria come mostrato nell'esempio, permette la definizione di vincoli personalizzati. In Coypu è presente una classe "Shows" che implementa diverse condizioni e aumenta la leggibilità delle asserzioni, come si può vedere in listato 4.22.

Listato 4.22: Uso del tipo "Shows" di Coypu.

---

```
1 Assert.That(footer, Shows.Css("img"));
```

---



## 4.4 Introduzione del CSS

I metodi presenti nella libreria di Coypu non includono le funzionalità per leggere le proprietà di stile degli elementi presenti nel DOM di una pagina.

Listato 4.23: Seconda funzionalità per CSBlog

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

Nella funzionalità corrente dovrebbero essere verificati alcuni scenari relativi all'aspetto della pagina, in particolare gli effetti cromatici descritti nei precedenti capitoli e altre proprietà legate alla corretta visualizzazione degli elementi.

Nonostante gli sforzi dell'autore e degli sviluppatori, che frequentemente propongono le loro pull-request su GitHub, Coypu non può mantenere il livello di sviluppo di Selenium. Per fornire comunque pieno supporto ai propri utenti è possibile sfruttare la libreria nativa di Selenium stesso, utilizzata dalle stesse funzionalità di Coypu, come mostrato nel passo del listato 4.24.

Listato 4.24: Uso delle funzionalità native di Selenium.

```
1 [Given(@"l'intestazione ha un colore di sfondo")]
2 public void DatoLIntestazioneHaUnColoreDiSfondo(){
3     var selenium = ((OpenQA.Selenium.Remote.RemoteWebDriver)browser.
4         Native);
5     var color = selenium.FindElementById("header").GetCssValue("
6         background-color");
7     Assert.NotNull(color);
8     Assert.AreEqual("rgba(46, 47, 48, 1)", color);
9 }
```

Nel metodo è descritto l'utilizzo delle funzionalità di Selenium ed in particolare l'invocazione del metodo "GetCssValue" per ottenere la rappresentazione "rgba" del colore di sfondo dell'intestazione.

La versione per C# di Selenium è estremamente simile a quella per Java, per quanto possibile compatibilmente con le diverse sintassi. Per questo motivo illustrare l'implementazione degli scenari per la funzionalità corrente, non aggiungerebbe ulteriori informazioni. Il materiale corrispondente è pertanto omissis.

## 4.5 Definizione del modello

In questa sezione sono descritte le componenti di MVC5 che permettono la gestione delle entità attraverso CSBlog e lo sviluppo dei relativi test di accettazione.

Listato 4.25: 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

```

---

### 4.5.1 Dipendenze

Anche per i test in CSBlog è necessario provvedere alla regressione del modello, rimuovendo dal database le entità create per verificare uno scenario. Come descritto nelle sezioni precedenti, con Specflow e Coypu sono stati definiti due hook per la creazione e rilascio della sessione del browser, che assicurano l'esecuzione dei test in nuove finestre e la rimozione di ogni cookie e sessione HTTP relativi a precedenti azioni.

Per completare il processo di regressione è necessario provvedere a rimuovere anche eventuali post creati negli scenari. La libreria di testing prevede l'utilizzo del tag “@clear” per individuare le funzionalità o gli scenari che introducono ma non eliminano nuovi contenuti sul blog.

Nel successivo listato è mostrato l'intera callback eseguita alla conclusione degli scenari, indipendentemente dal risultato delle asserzioni.

Listato 4.26: Callback relativa alla terminazione degli scenari.

---

```

1 [AfterScenario]
2 public void AfterScenario(){
3     if(ScenarioContext.Current.ScenarioInfo.Tags.Contains("clear") ||
       FeatureContext.Current.FeatureInfo.Tags.Contains("clear")){
4         login();
5         string LoremIpsumTitle = "Lorem Ipsum";
6         string xpathQuery = String.Format("//div[@class = 'post'] [p/a[
           contains(text(), '{0}')]"]", LoremIpsumTitle);
7         browser.Visit("/");
8         foreach (var post in browser.FindAllXPath(xpathQuery)) {
9             post.FindCss(".remove_post_button").Click();
10            browser.ClickButton("Confermi la rimozione?");
11        }
12    }
13    _browser.Dispose();
14 }

```

---

L'attributo “AfterScenario” accetta come parametri un numero variabile di stringhe rappresentanti i tag per specificare per quali scenari debba essere eseguito il metodo. In Cucumber l'ordine di esecuzione degli hook coincide con l'ordine di

registrazione dei metodi, mentre in Cucumber-JVM, che sfrutta il meccanismo delle annotazioni, è possibile definire un attributo per indicare la priorità di ciascuna callback.

Al contrario in Specflow, non è possibile specificare alcun ordinamento ed invece di definire due callback, rispettivamente per la rimozione di eventuali post presenti nel blog e la conclusione della sessione di test, è necessario all'interno di un generico hook verificare manualmente la presenza del tag “@clear” per la funzionalità o lo scenario corrente ed eventualmente effettuare la rimozione dei post.

Per implementare gli hook in SpecFlow è necessario utilizzare gli attributi per associare le callback agli eventi (ad esempio Before/After Test) e implementare i metodi all'interno di classi annotate con l'attributo “Binding”, come per la definizione delle classi contenenti i passi.

### 4.5.2 Gestione dei form

Nella sottosezione corrente è descritta l'implementazione delle componenti necessarie per l'implementare le funzionalità per la creazione di un nuovo post. Come specificato nell'introduzione, VS permette la generazione dei controlli, attraverso un wizard di configurazione, e delle viste corrispondenti.

Lo sviluppo si è svolto in maniera veloce, se paragonato a Spring, potendo sfruttare una struttura già esistente e dovendo esclusivamente personalizzarne il comportamento.

#### I controlli

Ad ogni componente è associato un controllo ed una collezione di azioni eseguibili; nei seguenti frammenti di codice sono mostrate le implementazione delle azioni per la creazione dei post.

Listato 4.27: Frammento di codice contenente l'azione “Create” per la visualizzazione della relativa vista.

```
1 public class PostController : Controller { private CSBlogEntities
    db = new CSBlogEntities();
2     public ActionResult Create(){
3         if (!IsAuthorized())
4             return RedirectToLoginPage();
5         return View();
6     }
7     /*...*/
```

Come in RoR, sono presenti due metodi “Create” per disaccoppiare la gestione della vista per la creazione di un nuovo post in listato 4.27 e la gestione della persistenza dell'oggetto definito attraverso l'interfaccia dell'applicazione nel modello, in listato 4.28.

La chiamata di “View”, senza specificare parametri risolve la richiesta visualizzando la vista “/Post/Create.cshtml”, il cui percorso è inferito a partire dall'identificatore dell'azione ed il nome dell'entità.

Listato 4.28: Frammento di codice contenente l'azione "Create" persistenza del nuovo oggetto creato.

---

```

1 // POST: /Post/Create
2 [HttpPost]
3 [ValidateAntiForgeryToken]
4 public ActionResult Create([Bind(Include = "title,body")] Post post
5 ){
6     if (!IsAuthorized())
7         return RedirectToLoginPage();
8     if (ModelState.IsValid){
9         post.id = Guid.NewGuid();
10        post.createdAt = post.updatedAt = DateTime.Now;
11        post.authorId = (Guid)Session["author_id"];
12        db.Posts.Add(post);
13        db.SaveChanges();
14        TempData["notice"] = String.Format("Il post '{0}' è stato
15        creato con successo.", post.title);
16        return RedirectToAction("Details", new { post.id });
17    }
18    return View(post);
19 }
```

---

Nonostante la creazione di nuovi post sia stata suddivisa in due azioni, il metodo "Create" in listato 4.28 è piuttosto articolato. Di seguito sono descritte i principali aspetti:

- l'espressione booleana "ModelState.IsValid" verifica che i vincoli definiti a livello di logica dell'entità siano soddisfatti e si possa procedere nell'esecuzione delle istruzioni per rendere persistenti i dati inseriti dall'utente;
- l'attributo "ValidateAntiForgeryToken" protegge l'applicazione, ed in particolare le pagine che permettono la definizione di nuovi contenuti, da attacchi di over-posting e dalla forgiatura di richieste cross-site. Il processo per implementare la sicurezza dei form è abbastanza semplice e consiste nella generazione di un valore unico all'interno del form durante la visita della pagina e la copia del valore all'interno di un cookie HTTP creato ad-hoc; al momento dell'invio dei dati inseriti, MVC5 verifica l'uguaglianza dei due valori ed eventualmente permettendo l'operazione;
- l'attributo "Bind" del parametro "post" specifica quali campi del form devono essere utilizzati per inizializzare l'oggetto, gli attributi della classe "Post" comprendono anche altre informazioni che sono però inizializzate secondo la logica dell'applicazione;
- la proprietà "Session" permette l'accesso ai valori presenti nella corrente sessione HTTP, in cui è mantenuto un riferimento all'identificatore dell'utente che ha effettuato l'autenticazione;
- "db.Posts.Add" e "db.SaveChanges" rispettivamente aggiungono l'oggetto al modello ed effettuano la persistenza delle modifiche.

- il metodo “RedirectToAction” restituisce un’istanza di tipo “RedirectToRouteResult” per effettuare la ridirezione della richiesta dopo aver creato il post. Combinando l’uso della ridirezione con il dizionario “TempData” è possibile fornire alla vista, che verrà visualizzata al termine dell’azione corrente, delle espressioni il cui valore sarà mantenuto per una sola richiesta HTTP all’interno della sessione;
- Nel caso in cui la validazione degli attributi del post non sia andata a buon fine, il post restituisce un riferimento alla vista “Create” fornendo lo stesso parametro ricevuto in input per inizializzare i campi del form;

Come in Spring, in cui è possibile sfruttare i parametri di tipo “Model” per passare degli oggetti alla vista, in MVC5 è disponibile la proprietà “ViewBag” di tipo “dynamic” presente nella super classe “Controller”. Il tipo “dynamic” in C# posticipa le operazioni di type-checking normalmente eseguite a compile time al momento dell’esecuzione, semplificando l’utilizzo delle proprietà come in questo caso.

Listato 4.29: Esempio di utilizzo della proprietà “ViewBag”.

---

```
1 ViewBag.message = "Hello ViewBag";
```

---

Nel listato 4.29 è inizializzato con una stringa l’attributo dinamico “message”, che sarà utilizzabile nelle viste in Razor.

## Le viste

Dopo aver mostrato i controlli per la creazione di nuovi post, è ora descritta la vista per la creazione di nuovi post in listato 4.30.

Listato 4.30: Vista parziale relativa al form per la creazione dei post.

---

```
1 @model Blog.Models.Post
2 @{
3     ViewBag.PageTitle = ViewBag.Title = "Crea un post";
4 }
5 @using (Html.BeginForm()){
6     @Html.AntiForgeryToken()
7     <div class="form-horizontal">
8         @Html.ValidationSummary(true)
9         <div class="form-actions">
10             <input alt="Scrivi un nuovo post" id="submit" src="@Url.Content
              (~/Images/save_48.png)" type="image"
              value="Create" />
11         </div>
12         <p id="form_title">
13             Scrivi un nuovo post
14         </p>
15         <div class="form-group">
16             @Html.LabelFor(model => model.title, new { @class = "control-
              label" })
17         <div>
18             @Html.EditorFor(model => model.title)
19             <div class="error_explanation">
20                 @Html.ValidationMessageFor(model => model.title)
```

```

21         </div>
22     </div>
23 </div>
24 <!-- .. -->

```

In MVC, tramite le funzionalità di Razor è possibile tipare le proprie viste indicando, tramite la sintassi “@model”, qual è l'entità rappresentata dalla pagina corrente. Nella classe “Controller” sono presenti le funzionalità per istanziare degli oggetti di tipo “ViewResult” fornendo un'istanza di “object” che rappresenterà il modello, come mostrato nel listato 4.28 in caso di fallimento della validazione.

Nella vista sono utilizzate diverse funzionalità di Razor:

- nella parte iniziale della vista sono inizializzati alcuni valori della proprietà “ViewBag” utilizzati all'interno della vista rappresentante il layout per impostare il titolo della pagina e dell'intestazione di CSBlog;
- il metodo “BeginForm”, della classe “FormExtensions” di Razor, genera il codice HTML relativo all'elemento “form” e permette di utilizzare altri metodi per la definizione delle varie componenti del form;
- il metodo “ValidationSummary” restituisce le informazioni sulla validazione, nel caso in cui la vista attuale sia visualizzata come conseguenza del fallimento delle operazioni di persistenza;
- i metodi “LabelFor”, “EditorFor” e “TextAreaFor” generano gli elementi di input del form e i relativi “label” HTML;
- il metodo “ValidationMessageFor” ritorna i messaggi d'errore che descrivono i problemi riscontrati durante la validazione dell'entità, opportunamente formattati in HTML;

### Gestione degli errori nei form

Utilizzando gli attributi dell'EF è possibile definire dei vincoli sugli attributi delle entità. Nel listato 4.31 è mostrato un frammento di codice relativo alla proprietà “title” dell'entità “Post”.

Listato 4.31: Proprietà “title” dell'entità Post.

```

1 [Required(ErrorMessage = "Titolo mancante.")]
2 [StringLength(100, MinimumLength = 5, ErrorMessage = "Il titolo
   deve essere compreso fra 5 e 100 caratteri.")]
3 [Remote("CheckForDuplication", "Post", AdditionalFields = "id")]
4 [Display(Name = "Titolo", Description = "Inserisci in questo campo
   il titolo che vuoi dare al tuo articolo.")]
5 public string title { get; set; }

```

Gli attributi “Required” e “StringLength” effettuano la verifica del valore della stringa e invalidano l'oggetto nel caso il titolo sia mancante, troppo corto o troppo lungo. E' anche possibile associare a ciascun attributo un messaggio d'errore personalizzato che rappresenta il testo mostrato all'interno della vista.

Per effettuare operazioni di validazione come la verifica d'unicità del titolo, non è possibile sfruttare gli attributi già definiti in EF. Per verificare la presenza di altri post con il titolo simile all'interno del blog è effettuata una richiesta HTTP asincrona tramite JQuery, definita tramite l'attributo "Remote" ed i parametri "CheckForDuplication", "Post" e "id" che rispettivamente rappresentano l'azione e il controllo che gestiranno la richiesta e il parametro aggiuntivo "id"<sup>4</sup>.

Nel listato 4.32, utilizzando LINQ per accedere al modello e JSon per definire il risultato dell'azione, è verificato in funzione dei parametri la presenza di eventuali post con titolo simili.

Listato 4.32: Azione per la verifica dell'unicità del titolo dei post.

```

1 [HttpGet]
2 public JsonResult CheckForDuplication(Guid? id, string title){
3     var post = db.Posts.Where(p => p.title.Equals(title,
4         StringComparison.CurrentCultureIgnoreCase)).FirstOrDefault();
5     if (post != null && (id == null || !post.id.Equals(id)))
6         return Json("Il titolo è già presente.", JsonRequestBehavior.
7             AllowGet);
8     return Json(true, JsonRequestBehavior.AllowGet);
9 }
```

L'azione restituisce un valore serializzato tramite JSon, una stringa contenente il messaggio in caso di fallimento della validazione, oppure il valore "true" se il titolo è utilizzabile.

### Implementazione dei test

Oltre alle funzionalità per la navigazione, la classe "BrowserSession" di Coypu fornisce le funzionalità per la gestione dei form e la selezione degli elementi del DOM in funzione del tipo e del testo mostrato.

Listato 4.33: Login su CSBlog tramite Coypu.

```

1 [Given(@"mi autentico come "(.*)")"]
2 [When(@"mi autentico come "(.*)")"]
3 public void DatoMiAutenticoCome(string email){
4     string password = "password";
5     browser.ClickLink("Login");
6     browser.FillIn("Email").With(email);
7     browser.FillIn("Password").With(password);
8     browser.ClickButton("Login");
9 }
```

Rispetto a Selenium, il DSL di Coypu è più semplice e facilmente utilizzabile; inoltre è evidente la propensione della libreria a favorire la definizione di selettori in funzione del testo mostrato dai vari elementi, come ad esempio nel listato 4.33.

Queste pratiche permettono la definizione di test leggibili e di semplice comprensione, semplificando la manutenzione del codice.

<sup>4</sup>La verifica dell'unicità del titolo del post è effettuata sia alla creazione che alla modifica di un post, il parametro aggiuntivo "id" può assumere valore nullo ed è utilizzato per associare il titolo ad un eventuale post esistente.

Listato 4.34: Definizione di un selettore tramite i metodi di Coypu.

---

```

1 protected ElementScope findPostByTitle(String title) {
2     return browser.FindAllCss(".post").First(p => p.FindLink(title,
        new Options { TextPrecision = TextPrecision.Substring }).
        Exists());
3 }

```

---

Il metodo del listato 4.34 definisce una funzionalità ausiliaria per ricercare all'interno del DOM un post, contenuto all'interno di un "div" con attributo di "class" uguale a "post", il cui titolo, rappresentato da un collegamento, contenga parte del testo indicato dal parametro.

Il metodo è una funzionalità utile per individuare il l'elemento HTML che contiene le informazioni visualizzate per un certo post, il cui risultato è utilizzato per effettuare ulteriori accessi all'HTML interno.

Nelle versioni in Ruby e Java, lo stesso metodo è stato definito utilizzando un selettore ed un'espressione XPath. Con Coypu e le funzionalità dell'interfaccia "IEnumerable", che permettono l'uso di funzioni, è possibile definire query articolate ma allo stesso tempo comprensibili. Inoltre la definizione di selettori attraverso una concatenazione di metodi riduce la fragilità dei metodi semplificando le manutenibilità del codice e permette la verifica durante la compilazione rispetto ad una query XPath rappresentata in una stringa.

### Selettori

In Coypu sono implementati una buona varietà di metodi per l'implementazione di funzionalità per la definizione di selettori ed in generale effettuare ricerche all'interno del DOM. Oltre i metodi specifici per un tipo di elemento, come "FindLink" già mostrato in precedenza, sono presenti anche funzionalità più simili a quelle viste per Selenium.

Di seguito sono proposti alcuni esempi e frammenti dei passi contenuti alcuni dei metodi esistenti nella libreria.

Listato 4.35: Esempio del metodo "FindCss".

---

```

1 browser.FindCss(".post_title a", text: title).Exists();

```

---

In maniera simile a Capybara, ed in particolare ai metodi del modulo "Finders", nel metodo "FindCSS" è possibile specificare delle stringhe o delle espressioni regolari oltre alle regole del CSS.

Listato 4.36: Esempio del metodo "FindId".

---

```

1 protected ElementScope header {get{return browser.FindId("header")
    ;}}

```

---

Oltre alle funzionalità classiche delle librerie di automazione web, come il metodo "FindId", Coypu cerca di fornire delle funzionalità ulteriori tramite parametri opzionali o implementando metodi derivati come "FindIdEndingWith".

Listato 4.37: Esempio del metodo "FindAllCss".



```

1 public void DatoIlPostNonEleggibileSuCSBlog(string title){
2     browser.Visit("/");
3     var posts = browser.FindAllCss(".post");
4     foreach (var post in posts) {
5         Assert.That(!post.FindLink(title).Exists());
6     }
7 }

```

---

A differenza di Capybara, Coypu non restituisce errore nel caso in cui venga utilizzato un metodo per individuare un singolo elemento ed al contrario siano presenti più potenziali risultati, ma restituisce il primo elemento incontrato.

Per gestire collezioni di elementi sono disponibili metodi come “FindAllCss”. Rispetto ai metodi che restituiscono un singolo risultato, i metodi che ricercano tutti i potenziali riscontri all'interno della pagina non applicano strategie di attesa per eventuali elementi asincroni, bensì restituiscono la collezione di elementi presenti al momento dell'invocazione; per modificare il comportamento è possibile definire un predicato per descrivere lo stato atteso.

Listato 4.38: Esempio del metodo “FindXPath”.

---

```

1 [Then(@"l_intestazione è posizionata all'inizio")]
2 public void AlloraLInizio(){
3     var first = browser.FindXPath("//body/*[1]");
4     Assert.True(first.Exists());
5     Assert.True(header.Exists());
6     Assert.AreEqual(first.Id, header.Id);
7 }

```

---

Nonostante la libreria propenda per l'utilizzo di funzionalità ad alto livello, per favorire la leggibilità dei test, è anche possibile definire i seletori tramite espressioni XPath, come in listato 4.38

Listato 4.39: Utilizzo della classe Options di Coypu.

---

```

1 protected ElementScope FindNotice(string noticeMessage){
2     var notice = browser.FindId("notice", new Options { TextPrecision
3         = TextPrecision.Exact});
4     Assert.That(notice.Exists());
5     return notice;
6 }

```

---

In alcune parti della libreria si può notare la forte ispirazione che Capybara ha dato al progetto, ad esempio i metodi permettono l'uso di un parametro “Options,” che ricorda l'uso del dizionario come parametro opzionale in Ruby, come in listato 4.39.

Le istanze della classe “Options” permettono di modificare il comportamento dei metodi, specificando la strategia di attesa e di polling, il tipo di confronto sulle stringhe da utilizzare, la cardinalità attesa e molti altri parametri.

### 4.5.3 Debug con Specflow

Durante lo sviluppo di CSBlog con VS sono state individuate alcune difficoltà nell'applicare la tecnica dell'ATDD a causa dell'impossibilità di eseguire il debug dei test. All'interno dell'ambiente di sviluppo è possibile eseguire la propria applicazione sia in modalità debug classica, in cui VS monitora l'esecuzione ed eventualmente gestisce e segnala gli errori, sia in una modalità debug semplificata in cui è possibile continuare a sviluppare nell'ambiente di sviluppo.

Dai tentativi fatti su VS, qualsiasi sia il tipo di esecuzione dell'applicazione sembra che non sia possibile eseguire i propri test di accettazione in modalità debug, perché è consentito che venga eseguito al più un solo processo per ogni istanza di VS.

Creando un secondo progetto contenente esclusivamente i test ed aprendo due istanze di VS, una per lo sviluppo ed una per il testing, si dovrebbero evitare conflitti durante l'esecuzione, introducendo però possibili rallentamenti nel sistema.

## 4.6 Login & Autorizzazione

Listato 4.40: Autenticazione in CSBlog.

---

```
1 @cap6
2 Funzionalità: Autenticazione su SBlog
3   Come Autore di SBlog
4   Vorrei che alcune operazioni sensibili siano permesse previa
      autenticazione
5   Per poter garantire l'autenticità dei contenuti
```

---

Come per le funzionalità riguardanti l'analisi del CSS e delle proprietà estetiche dei nodi dell'HTML, Coypu non implementa metodi specifici per la gestione dei cookie e delle sessioni.

E' comunque possibile utilizzare le funzionalità native per effettuare operazioni sui cookie e le sessioni HTTP come già descritto nel capitolo su SBlog.

### 4.6.1 Accesso ai singoli attributi

La classe "ElementScope" di Coypu ridefinisce il comportamento dell'operatore "[ ]": come in Capybara, è possibile accedere alle singole proprietà indicando tramite una stringa l'identificatore desiderato. Accedere alle proprietà di un elemento HTML come se fosse un array associativo presenta alcuni vantaggi: la sintassi è facilmente leggibile e comprensibile, inoltre, dal punto di vista dell'implementazione, è facilmente estendibile nel caso il W3C prevedesse delle variazioni nell'HTML, come è ad esempio accaduto per la versione HTML5.

## 4.7 Asincronia

Coypu gestisce gli elementi della pagina con lo stesso principio di Capybara: ogni elemento può potenzialmente essere asincrono, preferendo la definizione di una strategia di ricerca globale tramite attese e polling, rispetto alla definizione di asserzioni sullo stato degli elementi, perché difficili da esprimere, implementare e mantenere.

### 4.7.1 JavaScript

Listato 4.41: Introduzione di un breve script Javascript.

---

```

1 Funzionalità: Easter Egging
2   Come Sviluppatore
3   Vorrei che nel blog fosse presente un mio logo
4   Per firmare il mio lavoro
5
6   Contesto:
7   Dato apro CSBlog
8   Scenario: EasterEgg
9   Dato non è presente il logo nell'intestazione
10  Quando clicco sull'area del piè di pagina
11  Allora è presente il logo

```

---

Nei successi listati 4.42, 4.43 e 4.44 è mostrata l'implementazione dello scenario "EasterEgg". Rispetto alle attese esplicite di Selenium e a Capybara che sfrutta il metodo "synchronize" per verificare il completamento delle operazioni asincrone, l'implementazione è nettamente più compatta e leggibile; inoltre non è presente alcuna particolarità rispetto ai test descritti nelle precedenti sessioni che suggerisca che il piè di pagina implementi un comportamento asincrono.

Listato 4.42: Pre-condizione.

---

```

1 [Given(@"non è presente il logo nell'intestazione")]
2 public void DatoNonEPresenteIlLogoNellIntestazione(){
3     Assert.That(footer, Shows.No.Css("img"));
4 }

```

---

Listato 4.43: Evento.

---

```

1 [When(@"clicco sull'area del piè di pagina")]
2 public void QuandoCliccoSullAreaDelPieDiPagina(){
3     footer.Click();
4 }

```

---

Listato 4.44: Asserzione.

---

```

1 [Then(@"è presente il logo")]
2 public void AlloraEPresenteIlLogo(){
3     var footer = base.footer;
4     Assert.That(footer, Shows.Css("img"));
5     Assert.That(footer.FindId("woodstock").Exists());
6 }

```

---

### 4.7.2 Scenari sull'auto-completamento con JQuery UI

Listato 4.45: Scenario sulla ricerca nei post.

---

```

1 @cap5
2 Funzionalità: Ricerca fra i post
3   Come Lettore
4   Vorrei poter ricercare i post su RBlog
5   Per poter navigare fra i contenuti più velocemente

```

---

Nella funzionalità corrente è verificato il widget per l'auto-completamento del menu della ricerca, implementato tramite JQuery UI ed una chiamata AJAX.

Listato 4.46: Scenario riguardante l'auto-completamento della ricerca.

---

```

1 Scenario: Autocompletamento della ricerca
2   Dato nell'intestazione è presente la barra di ricerca
3   Dato il post "Lorem Ipsum" esiste
4   Quando inserisco il testo "lor" da ricercare
5   Allora viene proposto il post "Lorem Ipsum"
6   Quando inserisco il testo "xyz" da ricercare
7   Allora non è proposto alcun post
8   ...

```

---

Nei successivi listati 4.47 e 4.48 sono incluse le implementazioni dei passi che verificano la presenza del menù a tendina per il campo "input" della ricerca.

Listato 4.47: Verifica della presenza di un post nei suggerimenti.

---

```

1 [Then(@"viene proposto il post "(.*)""")]
2 public void AlloraVienePropostoIlPost(string title){
3     var suggestion = browser.FindCss(".ui-menu-item", title);
4     Assert.That(suggestion.Exists());
5 }

```

---

Coypu non necessita di introdurre istruzioni ad-hoc anche per uno scenario più complesso, quale questa operazione asincrona, che ha maggior latenza rispetto alla risoluzione della chiamata Javascript dello scenario "EasterEgg". Inoltre Coypu è coerente nell'implementazione delle proprie funzionalità e restituisce solo elementi effettivamente visibili nella pagina<sup>5</sup>. Al contrario utilizzando Selenium per questo stesso test è stato necessario verificare l'effettiva visualizzazione dell'elemento "ui-menu-item" tramite il metodo "isDisplayed" dell'interfaccia "WebElement" a causa del comportamento particolare del widget.

Listato 4.48: Verifica dell'assenza di suggerimenti.

---

```

1 [Then(@"non è proposto alcun post")]
2 public void AlloraNonEPropostoAlcunPost(){
3     Assert.That(browser, Shows.No.Css(".ui-menu-item"));
4 }

```

---

<sup>5</sup>Tramite la configurazione della sessione di navigazione, è possibile richiedere che siano individuati anche gli elementi non visibili.

Il widget per l'auto-completamento, probabilmente per minimizzare il numero di operazioni di modifica del DOM, applica un meccanismo paragonabile al caching. Invece di rimuovere le opzioni proposte dalla pagina, che corrispondono ai punti di una lista numerata in HTML, quando il focus è spostato su un altro elemento della pagina, JQuery UI rende non visibile l'intera lista e ne mantiene gli elementi nel DOM.

Quando sarà effettuata una nuova ricerca all'interno della pagina, il widget provvederà a correggere la cardinalità della lista e sostituire il testo degli elementi con i nuovi valori.

## Capitolo 5

# Conclusione

### 5.1 I framework

In questa prima sezione sono analizzati gli ambienti di sviluppo e le principali funzionalità dei framework utilizzati, ed effettuato un confronto.

#### 5.1.1 Produttività

Una delle caratteristiche fondamentali per valutare un framework e l'ambiente di sviluppo scelto è la produttività, intesa come semplicità di gestire il framework e le tecnologie necessarie per il progetto e l'immediatezza della configurazione.

#### IDE: usabilità, disponibilità e costi

Per sviluppare le applicazioni web descritte nella tesi sono stati scelti gli ambienti di sviluppo più utilizzati e meglio integrati con ciascuna tecnologia. Nonostante tutti permettano di implementare applicazioni web tramite i framework MVC documentati nei precedenti capitoli, i prodotti analizzati sono molto differenti fra loro per quanto riguarda il tipo di business, i prezzi e il processo di sviluppo.

Seguendo lo stesso ordine dei capitoli, il primo IDE utilizzato è RubyMine, un prodotto di JetBrains. Il software è sviluppato da un'azienda privata ed attualmente, secondo il loro sito, il prezzo parte da circa 100€ per una singola licenza ed arriva 200€ per una licenza commerciale, valida per aziende ed associazioni.

Lo sviluppo di RubyMine è costante con frequenti aggiornamenti. Nel corso della tesi sono stati installati diversi aggiornamenti gestiti internamente all'IDE, senza necessità di scaricare patch e dover riconfigurare il proprio ambiente di sviluppo.

La versione utilizzata per implementare RBlog è quella professionale, mediante una licenza accademica gratuita.

Spring Tool Suite, la versione di Eclipse specializzata per supportare lo sviluppo con Spring, è un progetto privato della "Pivotal Software", l'azienda che cura lo

sviluppo dei diversi prodotti per Spring. Come già descritto all'interno del relativo capitolo, solo il plugin è sviluppato dagli autori di Spring, mentre il resto dell'applicazione è una versione di Eclipse per JavaEE, uno dei diversi progetti open-source sviluppati interamente da una comunità di sviluppatori attiva e numerosa.

A differenza dell'ambiente di sviluppo, l'insieme di funzionalità che definiscono STS non è un progetto open-source, ma è comunque distribuito a titolo gratuito.

Per rilasciare aggiornamenti Eclipse e STS forniscono direttamente la nuova versione del prodotto, senza aggiornarne le funzionalità internamente. Questo processo può essere scomodo nel caso in cui sia necessario installare plugin specifici per lo sviluppo non inclusi di default nel software.

Visual Studio è un prodotto a pagamento sviluppato da Microsoft. Per lo sviluppo di CSBlog è stata utilizzata la versione Ultimate 2014, che rappresenta il prodotto più completo fra le varie versioni disponibili.

Come descritto nel relativo capitolo, VS è un prodotto estremamente professionale e versatile. Gli aggiornamenti sono gestiti internamente al software, mantenendo le impostazioni personali dell'utente.

La versione utilizzata per lo sviluppo è una licenza accademica ottenuta tramite la convenzione DreamSpark fra Microsoft ed Unige. Acquistare una copia di VS è decisamente oneroso, ma sono anche previste versioni essenziali come Visual Studio Express 2013, che è disponibile gratuitamente per scopi non commerciali.

Nonostante VS sia al momento eseguibile solo utilizzando un sistema operativo Windows, sia più esigente in termini di requisiti di sistema e sia a pagamento, ritengo che sia il prodotto più completo e professionale fra quelli utilizzati.

	Multipiattaforma	Costo	Versione Accademica/No Profit
RubyMine	✓	100€ - 200€	✓
STS	✓	Gratis	Non necessaria
VS	✓	Diversi piani d'acquisto, ma molto costosi	✓

Tabella 5.1: Valutazione riassuntiva degli IDE

	Facilità di		
	Installazione	Aggiornamento	Uso
RubyMine	+	+	+
STS	+	Sconsigliato	+/-
VS	+	+	+

Tabella 5.2: Valutazione riassuntiva dell'esperienza di sviluppo



### Supporto a sviluppo di applicazioni MVC

Implementare un'applicazione web tramite un framework MVC comporta l'utilizzo di un vasto stack di strumenti e tecnologie.

RubyMine è un ambiente di sviluppo specifico per implementare applicazioni RoR ed include il supporto a tutti i linguaggi comunemente utilizzati in ambito web ed anche a nuove tecnologie<sup>1</sup>.

Oltre a fornire gli strumenti per supportare l'implementazione delle applicazioni, RubyMine integra la possibilità di eseguire test di diversa granularità senza dover installare e configurare plugin. Per lo sviluppo di RBlog e l'implementazione della tecnica dell'ATDD è stata utilizzata la versione classica dell'IDE senza installare alcuna componente aggiuntiva, anche se l'ambiente di sviluppo fornisce gli strumenti per installare funzionalità aggiuntive.

STS si basa sulla versione per Java EE di Eclipse, che anche nella versione classica include molte funzionalità specifiche per lo sviluppo web. Per implementare e testare SBlog è stato necessario installare il plugin per l'integrazione di Thymeleaf e di Cucumber, tramite l'apposito Marketplace.

Inoltre Spring è un framework estremamente modulare, all'interno del quale lo sviluppatore ha la libertà di scegliere fra diversi strumenti per uno stesso scopo. Probabilmente a causa della natura open-source dell'IDE, i plugin presentano spesso errori nell'integrazione con l'ambiente di sviluppo e sono carenti di funzionalità non essenziali ma utili, come l'estensione di Cucumber-JVM, decisamente inferiore alla versione per SpecFlow in VS.

A differenza di RubyMine e STS, VS è un IDE non specializzato per lo sviluppo di applicazioni Web, ma genericamente utilizzato per tutte gli strumenti di .NET. Nonostante lo sviluppo tramite MVC5 non sia l'unica prerogativa dell'IDE, VS è definito in maniera modulare e procede all'installazione di componenti aggiuntive se e quando l'utente crea progetti che necessitano di estensioni rispetto alla versione di base del software, come l'integrazione con Sass.

Per supportare lo sviluppo di CSBlog e dei test è stato necessario installare tramite NuGet, i plugin per Coypu, PhantomJS e SpecFlow.

Come descritto nel capitolo su MVC5, utilizzando VS sono presenti delle difficoltà nell'esecuzione dei test di accettazione a causa dell'impossibilità di eseguire più di un processo. Questo complica l'uso dei test in modalità di debug. Al contrario, RubyMine è meno rigido e permette di monitorare il debug di un numero arbitrario di processi, che è una caratteristica utile per effettuare test che richiedono l'esecuzione del server web e dell'applicazione: ad esempio è una pratica comune effettuare stress test per ottenere indicazioni sulle performance, utilizzando strumenti come Gatling~[72].

---

<sup>1</sup>Fra i linguaggi supportati spicca CoffeeScript che permette di scrivere codice asincrono utilizzando una sintassi meno verbosa e generare codice JavaScript come risultato della compilazione.

Inoltre, configurando opportunamente il proprio progetto, in RubyMine è anche possibile eseguire in parallelo l'esecuzione dei test di accettazione~[70], permettendo anche di ottenere una valutazione indicativa delle performance di una singola funzionalità.

	Necessità di personalizzazione		Esecuzione in debug dei test
	MVC	ATDD	
RubyMine	✓	✗	✓
STS	✓	✓	✓
VS	✗	✓	✗

Tabella 5.3: Valutazione riassuntiva del supporto allo sviluppo

### Configurazione di un progetto

Uno dei fattori che incide sensibilmente nello sviluppo di una nuova applicazione MVC è la semplicità di configurare un nuovo progetto tramite l'ambiente di sviluppo.

In RubyMine la prima generazione di una nuova applicazione è effettuata attraverso un wizard. Al termine della procedura l'utente può già eseguire la propria applicazione, che è definita da un'insieme di viste -già strutturate utilizzando un layout- che contengono informazioni e guide su come iniziare lo sviluppo.

Per definire le componenti del proprio progetto è possibile utilizzare tramite l'IDE le funzionalità a riga di comando di RoR -anche se è necessario specificare le opzioni in maniera simile all'esecuzione tramite console- che generano la struttura delle componenti e forniscono all'utente una prima versione da personalizzare.

Per definire le dipendenze del proprio progetto e le librerie utilizzate è molto conveniente utilizzare il Gemfile, perché definisce un DSL essenziale e facilmente leggibile. Per la configurazione di proprietà specifiche dell'applicazione, come i parametri per connessione al database, sono utilizzabili script in YAML o in Ruby, generati durante la creazione del progetto.

La struttura di un progetto in Spring varia in funzione del build manager utilizzato, scegliendo ad esempio Gradle o Maven. Per comodità in SBlog è stato utilizzato Maven, perché già integrato nella versione di Eclipse utilizzata da STS.

Come per RubyMine o VS è prevista la possibilità di creare un nuovo progetto utilizzando Spring e selezionando quali funzionalità siano richieste nell'applicazione. Il wizard termina con la generazione di un progetto contenente esclusivamente la struttura dei package, che rispecchia l'organizzazione di Maven in file sorgenti, test e risorse, ed il file "pom.xml" per la configurazione delle dipendenze.

Quindi, scegliendo di generare un progetto che utilizzi JPA e Thymeleaf, all'esecuzione dell'applicazione sono sollevate diverse eccezione che segnalano l'assenza dei bean di configurazione del modello e la mancanza del provider per JPA.

Oltre alla generazione automatica dei progetti, sono presenti anche diversi progetti starter da cui prendere spunto, ma che non possono essere utilizzati come base del progetto in quanto sono specifici per una singola tecnologia, ad esempio sono presenti gli starter per JSP, Thymeleaf e JPA, ma non per combinazioni di tecnologie.

Per configurazione le singole proprietà di un'applicazione web è possibile combinare l'uso Spring Boot ed i beans. Nonostante includere fra le dipendenze di Maven i diversi progetti starter riduca il numero di aspetti da configurare, è comunque necessario definire alcuni bean, procedura che non è sempre intuitiva e rapida.

VS è l'ambiente di sviluppo più completo e user-friendly fra quelli utilizzati nella tesi. La generazione dei progetti e delle singole componenti è possibile tramite interfacce grafiche che guidano l'utente e documentano ogni passaggio attraverso esempi e spiegazioni.

Inoltre la creazione di un nuovo progetto genera un'applicazione completa e funzionante, anche se priva del modello.

## Conclusioni

	Generazione di progetto "completo"	Supporto alla configurazione
RubyMine	✓	+
STS	✗	-
VS	✓	++

Tabella 5.4: Valutazione riassuntiva del supporto allo sviluppo

Complessivamente, è risultato evidente come VS sia fortemente orientato alla produttività grazie all'ottima organizzazione e del continuo supporto fornito all'utente, rappresentando in assoluto il miglior IDE utilizzato nella tesi.

Anche RubyMine è un ambiente di sviluppo completo e ricco di funzionalità, ma rispetto a VS, manca del supporto costante e dei wizard per la generazione di componenti.

STS invece è l'IDE più carente in termini di funzionalità che esulino dal semplice sviluppo.

### 5.1.2 Il modello

#### Configurazione

La configurazione delle entità per il modello è uno degli aspetti più sensibili dell'intero sviluppo dell'applicazione MVC, in quanto ogni framework ha una propria interpretazione del modello, che differisce sia nei dettagli implementativi che più ad

alto livello nella rappresentazione delle entità.

La generazione delle entità in RoR è stata eseguita principalmente tramite il comando “scaffold”, che attraverso una sintassi intuitiva definisce le migrazioni per il modello, le classi rappresentanti le entità, i controlli e le azioni.

Per personalizzare il dominio del modello è sufficiente modificare il contenuto delle migrazioni e modificare le classi rappresentanti le entità attraverso vincoli e relazioni.

Listato 5.1: Frammento della classe Post di RBlog.

---

```

1 class Post < ActiveRecord::Base
2   belongs_to :author
3   validates :title, presence: {message: 'Titolo mancante'}
4   validates :title, length: {minimum: 5,
5     too_short: "Il titolo deve essere almeno di 5 caratteri",
6     maximum: 100,
7     too_long: "Il titolo deve essere al più di
8       100 caratteri"},
9   uniqueness: {
10     case_sensitive: false,
11     message: 'Esiste già un post con questo titolo' }
12   #..

```

---

Sfruttando ampiamente la sintassi di Ruby, le entità sono rappresentate attraverso una notazione versatile, estremamente leggibile e mantenibile. Rispetto a Spring e VS, le classi “Author” e “Post” sono le più compatte, misurando le linee di codice dei modelli dei tre progetti, ma anche le più espressive.

Al contrario, configurare il modello con Spring è un'operazione che può risultare complicata. Innanzitutto, rispetto a RoR che propone un'unica libreria per definire il modello e su questa definisce i propri strumenti e convenzioni, JPA è un progetto fortemente modulare con varie possibili implementazioni dei singoli moduli. L'utente deve quindi, ancor prima di implementare le classi, scegliere il provider che intende utilizzare (e verificarne le particolarità, ad esempio per EclipseLink è opportuno comprendere com'è compiuto il caricamento dei dati dai sistemi di persistenza).

Listato 5.2: Frammento dell'entità Post in JPA.

---

```

1 @Component
2 @Entity
3 public class Post {
4   /*...*/
5   @Id
6   @GeneratedValue(strategy = GenerationType.TABLE)
7   Integer id;
8
9   @ManyToOne(cascade = {CascadeType.MERGE, CascadeType.REFRESH},
10     fetch=FetchType.LAZY)
11   @JoinTable(name = AUTHOR_POST_JOIN_TABLE,
12     joinColumns = @JoinColumn(name = POST_JOIN_COLUMN),
13     inverseJoinColumns = @JoinColumn(name =
14       AUTHOR_JOIN_COLUMN), uniqueConstraints =

```

---

```

12      @UniqueConstraint(columnNames = {
          AUTHOR_JOIN_COLUMN, POST_JOIN_COLUMN}))
    Author author;

```

Dopo aver configurato il provider, definendo come per SBlog gli opportuni bean, è necessario comprendere il funzionamento di JPA e delle annotazioni per poter rappresentare correttamente i POJO sul database.

Per dare un'idea della complessità di JPA, nel tempo impiegato per definire il funzionamento dell'architettura di CSBlog, due/tre giorni uomo, è stato sviluppato il modello con Spring, risolti i diversi errori di configurazione e di caricamento lazy.

Per definire il modello per MVC5 è stata utilizzata la generazione del modello in funzione dello schema di un database SQL esistente. Le modifiche effettuate sulle classi parziali rappresentanti le entità hanno riguardato esclusivamente la definizione di metodi ausiliari e l'utilizzo di attributi per definire la validazione delle proprietà.

Listato 5.3: Frammento dell'entità Post con l'Entity Framework.

```

1 public partial class Post{
2     public System.Guid id { get; set; }
3     public System.Guid authorId { get; set; }
4     [Required(ErrorMessage = "Titolo mancante.")]
5     [StringLength(100, MinimumLength = 5, ErrorMessage = "Il titolo
        deve essere compreso fra 5 e 100 caratteri.")]
6     [Remote("CheckForDuplication", "Post", AdditionalFields = "id")]
        [Display(Name = "Titolo", Description = "Inserisci in
            questo campo il titolo che vuoi dare al tuo articolo.")]
7     public string title { get; set; }

```

Le classi "Post" ed "Author" sono paragonabili alle entità di RoR, ma meno espressive e leggibili, a causa della sintassi del C#, decisamente più verbosa di Ruby.

	Configurazione del sistema (DB)	Supporto per la definizione E/R	Qualità delle classi prodotte.
RubyMine	✓	✓	Leggibili e compatte
STS	✗	✓	Basso
VS	✓	✓	Leggibili e compatte

Tabella 5.5: Valutazione riassuntiva degli strumenti per la configurazione del modello

### Astrazione

Tutti i framework utilizzano la tecnica ORM per rappresentare le entry presenti nei database ed includere un livello di astrazione per semplificare la gestione del modello.

Le funzionalità per la persistenza -creazione, modifica e salvataggio- delle entità sono praticamente equivalenti; sono però presenti sensibili differenze nelle librerie

per l'implementazione di query.

Listato 5.4: Selezione dei titoli simili in RoR.

```
1 post_titles = Post.select(:title)
2   .where('lower(title) LIKE ?', "%#{params[:title]}%")
3   .order('created_at DESC')
4   .limit(10);
```

Sia in RoR che in MVC5, rispettivamente con l'interfaccia per le query degli Active Record e LINQ, è possibile definire delle interrogazioni utilizzando metodi e costrutti del linguaggio, mantenendo le operazioni consistenti con il livello logico. LINQ, essendo implementato in C#, può verificare a compile-time i tipi ed in generale è utilizzabile anche in contesti non correlati alla persistenza ed è più completo nelle funzionalità rispetto alla controparte per RoR.

Listato 5.5: Selezione dei titoli simili in LINQ.

```
1 var postTitles = db.Posts
2   .Where(p => p.title.ToLower().Contains(title.ToLower()))
3   .OrderByDescending(p => p.title)
4   .Select(p => p.title)
5   .Take(10)
6   .ToList();
```

In Spring è possibile definire le proprie query con le interfacce “Repository” e la libreria JPA Criteria per la generazione dinamica delle interrogazioni JPQL. Nonostante sia un metodo utile per implementare le interrogazioni più semplici, aumentando di complessità risulta poco scalabile, difficilmente mantenibile e leggibile. Inoltre non sono presenti molte funzionalità di SQL.

Listato 5.6: Selezione dei post in JPA.

```
1 public List<Post> findPostByTitleContainingIgnoreCase(String title)
2   ;
```

E' anche possibile definire le proprie query direttamente in JPQL, introducendo però una sintassi e delle meccaniche più simili a SQL che al paradigma ad oggetti.

	Persistenza	Query			
		Evitare	Controllo	Leggibilità	Compattezza
		linguaggi esterni	Statico		
RoR	ORM Classico	✓	✗	✓	✓
Spring	ORM Classico	Solo banali	✓	✗	✓
MVC5	ORM Classico	✓	✓	✓	-

Tabella 5.6: Valutazione riassuntiva dell'astrazione del modello e delle librerie per le interrogazioni

### Validazione

L'interpretazione della validazione delle entità è un altro aspetto per il quale sono presenti tre differenti interpretazioni.

In RoR lo schema del modello è gestito quasi interamente a livello logico, nel database è solo utilizzato il vincolo “not null” per le chiavi primarie. Astraendo in maniera così decisa la rappresentazione delle entità è possibile scegliere la miglior strategia di validazione ad alto livello, eseguendo le operazioni una sola volta, senza dover replicare a livello fisico tutte le verifiche. Il framework si fa carico dell'intera gestione e permette di definire condizioni che verifichino lo stato dell'intero dominio, come l'unicità del titolo.

Spring invece mantiene un approccio il più modulare possibile, fornendo all'utente le annotazioni di JPA per la validazione a basso livello e annotazioni del package “javax.validation.constraints” per eseguire controlli simili anche a livello logico, con le quali, non mantenendo il controllo dell'intero dominio, non è possibile però effettuare dei controlli in funzione delle entità già esistenti.

L'EF di MVC5 mantiene una via intermedia fra le due implementazioni descritte. Per validare un'entità è possibile utilizzare gli attributi disponibili, che eseguiranno l'operazione di validazione sia a livello logico sia sul database. Il caso di CSBlog è particolare rispetto alle altre due implementazioni, in quanto i vincoli sul database, la rappresentazioni delle chiavi esterne e primarie e le operazioni di propagazione sono stati definiti prima del modello dell'applicazione.

Fra i metodi discussi ognuno ha pregi e difetti:

- RoR definisce un database con uno schema estremamente fragile, assumendo di esserne l'unico utilizzatore, ma nasconde all'utente i dettagli implementativi e definisce un'unica interfaccia per la validazione;
- JPA non effettua alcuna assunzione e si adatta alle necessità dell'utente ma obbliga all'introduzione di ulteriori annotazioni, aumentando la complessità delle classi che rappresentano le entità;
- EF semplifica la definizione delle entità, non effettua assunzioni sul database ma duplica le operazioni di verifica, introducendo potenziali overhead.

### Conclusioni

Complessivamente, la gestione del modello (creazione, CRUD, vincoli e query) in Spring è complicata e poco efficace, probabilmente a fronte della grande libertà di scelta per la sua realizzazione.

Il modello in RoR è compatto e il DB generato ha uno schema leggero, mancando dei vincoli. I modelli in MVC5 risentono, a livello di rappresentazione, della maggiore prolissità di C#, ma godono dei maggiori controlli statici e generano un DB più sicuro, grazie all'inserimento dei vincoli, ma più pesante.

Pertanto, RoR risulta la scelta vincente, dal punto di vista del modello, quando gli sviluppatori sono molto competenti ed il DB è totalmente incapsulato dall'applicazione. Viceversa, MVC5 (EF) risulta vincente, sempre dal punto di vista del modello, quando il DB sia condiviso (ad esempio l'applicazione vada ad insistere sul DB aziendale usato anche da altri applicativi) o gli sviluppatori meno capaci.

### 5.1.3 I controlli

#### Configurazione

Anche per la definizione dell'instradamento delle richieste le differenze d'implementazione dei diversi framework sono sensibili.

RoR, rispetto all'interpretazione del modello, fornisce diverse soluzioni per risolvere l'instradamento delle richieste HTTP. Innanzitutto esiste una convenzione “/Entità/azione?parametri” per la risoluzione automatica, inoltre è anche possibile utilizzare il file “routes.rb” per impostare manualmente l'associazione di un'azione con un particolare URL ed anche definire la gerarchia delle entità per implementare un'architettura REST.

Listato 5.7: Instradamento delle pagine statiche di RBlog.

---

```

1 Rails.application.routes.draw do
2   root :to => 'pages#index', :as => 'root' get '/author' => 'pages
   #author'
3   get '/abstract' => 'pages#abstract'
4   get '/log_in' => 'sessions#new', :as => 'log_in'
5   get '/log_out' => 'sessions#destroy', :as => 'log_out'
6   #..

```

---

Nel frammento 5.7 è definito l'instradamento delle richieste alle pagine statiche e definito un simbolo per semplificare la definizione di collegamenti e azioni che sfruttano la ridirezione.

Anche MVC5 prevede l'utilizzo di convenzioni per la risoluzione dell'instradamento e la possibilità di definire il proprio pattern, come illustrato in listato 5.8.

Listato 5.8: per la risoluzione dell'instradamento in CSBlog.

---

```

1 routes.MapRoute(
2   name: "Default",
3   url: "{controller}/{action}/{id}",
4   defaults: new { controller = "Post", action = "Index", id =
      UrlParameter.Optional }

```

---

Al contrario l'instradamento in Spring non è risolto in funzione di convenzioni esistenti né è gestito in maniera separata dai controlli, ma è implementato tramite apposite annotazioni all'interno dei singoli controlli.



Complessivamente, sia RoR che MVC5 propongono soluzioni valide: le funzionalità in Ruby forniscono più libertà all'utilizzatore e maggior potere espressivo, mentre in MVC5 è possibile configurare il pattern globale dell'applicazione invece che utilizzare un metodo imposto. Spring risulta, invece, macchinoso da usare e il codice prodotto non isola correttamente l'aspetto di instradamento.

### Implementazione

In RoR esistono numerose convenzioni che minimizzano l'implementazione dei controlli minimali. Nel listato 5.9 è evidente come sia conveniente per l'utente aderire alle pratiche esistenti (addirittura l'attributo "@page\_title" è definito solo per comodità).

Listato 5.9: Visualizzazione di un singolo post in RBlog.

---

```

1 def show
2   @page_title = @post.title
3 end

```

---

Anche in MVC5 la possibilità di utilizzare un pattern per l'instradamento semplifica la definizione delle azioni, richiedendo di implementare esclusivamente la logica dell'operazione senza dover configurare eccessivamente aspetti legati al protocollo HTTP.

Rispetto all'implementazione di RoR, nell'azione di MVC5 del listato 5.10 è necessario gestire esplicitamente il parametro "id" e far corrispondere a ciascun errore uno stato HTTP per la risposta.

Listato 5.10: Visualizzazione di un singolo post in CSBlog.

---

```

1 public ActionResult Details(Guid? id) {
2   if (id == null){
3     return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
4   }
5   Post post = db.Posts.Find(id);
6   if (post == null){
7     return HttpNotFound();
8   }
9   return View(post);
10 }

```

---

Spring anche per i controlli non specifica alcuna convenzione ed è quindi necessario dichiarare ogni parametro tramite annotazioni, articolate e poco leggibili, che appesantiscono notevolmente la definizione del metodo. Inoltre la necessità di utilizzare parametri di output come Model, complica ulteriormente l'implementazione.

Listato 5.11: Visualizzazione di un singolo post in SBlog.

---

```

1 @RequestMapping(value =("/{id}", method = RequestMethod.GET)
2 public String show(@PathVariable Integer id, Model model) {
3   Post post = postService.getPost(id);
4   model.addAttribute("page_title", post.getTitle());
5   model.addAttribute("content_template", "/posts/show");

```

---

```
6     model.addAttribute("posts", new Post[] { post });
7     return super.defaultMapping(model);
8 }
```

---

L'unico vantaggio di Spring su MVC5 è di permettere l'implementazione di un meccanismo esterno al controllo per associare a ciascuna eccezione un corrispondente stato HTTP.

### Conclusioni

Dal punto di vista dei controlli, RoR risulta chiaramente il migliore, supportando una chiara divisione dei diversi aspetti e permettendo di scrivere codice molto compatto, grazie all'uso di convenzioni.

La totale mancanza di convenzioni in Spring, lo relega, dal punto di vista dei controlli, alla posizione di coda, mentre MVC5, pur essendo più prolisso di RoR e non forzando una specifica strutturazione con metodi di servizio risulta poco efficace.

### Helper e Service

Nell'interpretazione del pattern MVC, RoR e Spring prevedono un componente aggiuntivo rispetto a quelli previsti.

RoR fa ampio uso degli Helper, implementati in Ruby come moduli, per organizzare metodi di utilità e definire un'applicazione più lineare e fattorizzata. Essendo privi di stato, i moduli sono utilizzabili da qualsiasi componente senza dover applicare pattern come la DI.

In Spring, oltre ai ruoli “@Repository” e “@Entity” per il modello e a “@Controller”, è presente la componente “@Service”. I servizi sono utilizzati per fattorizzare la logica dell'accesso al modello da parte dei controlli e semplificarne le azioni, e istanziati tramite la DI e l'annotazione “@Autowired”.

In MVC5 non è prevista alcuna componente per fattorizzare metodi d'utilità o per introdurre un'astrazione ulteriore del modello; è quindi compito dell'utente definire un proprio pattern ed implementarlo.

#### 5.1.4 Le viste

Per implementare le viste dei blog sono stati utilizzati due tipologie di template engine: ERB e Razor definiscono il comportamento dinamico delle viste introducendo all'interno delle pagine HTML brevi script, definiti tramite elementi sintattici non analizzabili da un comune browser, Thymeleaf invece estende le proprietà dei comuni elementi HTML tramite ulteriori attributi i cui valori possono essere espressioni dinamiche.

---

Listato 5.12: Titolo di un post con Thymeleaf.

```
1 <p class="post_title">
```

```
2 <a th:href="/posts/${post.id}" th:text="${post.title}">
3 </a>
4 </p>
```

---

Dal punto di vista teorico del pattern MVC, le viste non dovrebbero contenere logica ed essere esclusivamente una rappresentazione delle entità presenti nel modello.

Thymeleaf definisce una grande varietà di attributi, il cui uso è orientato principalmente alla lettura ed interpretazione delle espressioni, semplificando lo sviluppo di viste che siano il più lineari possibili, ma sposta la complessità nelle azioni dei controller che devono fornire più parametri.

Al contrario le viste definite con ERB e Razor possono favorire la violazione del pattern introducendo all'interno dei documenti HTML dichiarazioni di variabili e facendo ampio uso di costrutti sintattici come cicli “for” o “if-else” per l'implementazione di logica e non solo a supporto della visualizzazione.

Potendo posticipare alla generazione delle viste una piccola porzione della logica dell'applicazione, le azioni di RoR e MVC5 spesso forniscono esclusivamente l'oggetto che è rappresentato nella vista, come il singolo post di cui mostrare gli attributi, negli esempi in listato 5.12 e 5.13.

---

Listato 5.13: Titolo di un post con ERB.

---

```
1 <p class='post_title'>
2 <%= link_to @post.title, @post %>
3 </p>
```

---

Personalmente trovo che la sintassi da Thymeleaf sia poco intuitiva e troppo articolata, preferisco quindi un approccio che permetta di scrivere nel linguaggio in cui è scritta l'applicazione perché più familiare e leggibile.

---

Listato 5.14: Titolo di un post con Razor.

---

```
1 <p class="post_title">
2 @Html.ActionLink(Model.title, "Details", new { id = Model.id })
3 </p>
```

---

Inoltre Thymeleaf, introducendo le espressioni all'interno degli attributi, e quindi come stringhe, non fornisce alcuna validazione a compile-time. Tutta la gestione degli errori è effettuata a run-time tramite eccezioni. Se per evitare di anticipare troppa logica nelle azioni si sceglie di implementare un'espressione articolata in Thymeleaf, è possibile trovarsi nella scomoda posizione di dover decifrare lo stack-trace delle eccezioni per bilanciare una parentesi.

Al contrario utilizzando un template engine come ERB o Razor, è possibile effettuare il debug della vista, utilizzare l'auto-complemento o l'IntelliSense e navigare fra i riferimenti per approfondire il funzionamento di un metodo ausiliario della libreria.

**Conclusioni**

Dovendo indicare il miglior template engine, credo che la sintassi di Ruby sia particolarmente adatta per operazioni di scripting come la definizione di viste dinamiche, inoltre gli helper utilizzabili in ERB hanno un'interfaccia più leggibile e sono meglio documentati delle funzionalità di Razor.

La difficoltà di uso di Thymeleaf lo rendono poco produttivo e quindi la scelta meno desiderabile, nonostante sia il più aderente al modello teorico.

## 5.2 ATDD

In questa sezione si confrontano i tre framework dal punto di vista del supporto fornito alla definizione, implementazione ed esecuzione dei test di accettazione.

### 5.2.1 BDD

Nel corso della tesi sono stati utilizzati tre framework per il BDD: le versioni per Ruby e Java di Cucumber e SpecFlow per .NET.

Tutti si basano su Gherkin, il DSL per la definizione dei test leggibili dagli utenti finali e strutturati in passi implementati indipendentemente. Il linguaggio per sviluppare i test di accettazione è ricco di funzionalità, ad esempio è possibile definire dei contesti comuni per le funzionalità, etichette per poter definire delle categorie di scenari o richiedere l'esecuzione di particolari callback, parametrizzare i test tramite tabelle e molto altro.

L'unico elemento sintattico che stona con la definizione di DSL Business Readable è l'uso dei tag, che introducono una notazione tecnica, la cui leggibilità dipende dallo sviluppatore.

Attualmente, Cucumber è il framework per il BDD più diffuso ed utilizzato. Cucumber nella versione originale per Ruby, e Cucumber-JVM sono praticamente equivalenti nell'implementazione del framework, anche se sfruttano elementi sintattici propri dei rispettivi linguaggi.

SpecFlow dà un'interpretazione più rigida del processo di associazione passo - implementazione, richiedendo la consistenza del tipo utilizzato ed introducendo il meccanismo degli "Scope" per poter definire più versioni di un stesso passo.

Rispetto a Cucumber è però carente nella definizione degli hook, essendo privo di un meccanismo per impostare un ordinamento nelle callback applicate allo stesso evento ed etichettate nella stessa maniera.

I framework risultano sostanzialmente equivalenti ed il fattore discriminante della scelta coincide quasi esclusivamente con il linguaggio in cui sono implementati. Se ci fosse la possibilità di scegliere liberamente fra le tre versioni, Cucumber è apparso più lineare nella gestione degli hook e quindi preferibile per il BDD dei blog, i cui test di accettazione non espongono diversi ruoli per gli utenti e mantengono una prospettiva comune dell'uso del sito.

Dovendo invece scegliere in funzione del supporto dato dai plugin nei diversi ambienti di sviluppo, la versione per RubyMine e quella per VS si equivalgono e sono entrambe valide, al contrario la versione per Eclipse implementa solo alcune funzionalità elementari, ma è comunque funzionante.

	Gherking	Passi Tipati	Ordinamento Hook	Plugin
Cucumber	Completo	✗	✓	Completo
Cucumber JVM	Completo	✗	✓	Parziale
SpecFlow	Completo	✓	✗	Completo

Tabella 5.7: Confronto riassuntivo fra i framework per il BDD.

### 5.2.2 Web Automation

Prima di analizzare le librerie degli strumenti utilizzati per automatizzare la navigazione web, è opportuno fare una precisazione sulla natura dei loro progetti.

Capybara e Coypu sono progetti open-source, disponibili su GitHub e mantenuti grazie agli sforzi degli autori e dei singoli sviluppatori che individuano bug ed effettuano correzioni. Entrambe i progetti sono attivi, anche se la comunità di sviluppatori di Coypu è più piccola e meno partecipe rispetto a quella di Capybara, ed esistenti da diversi anni, Capybara nasce nel 2009 e Coypu nel 2010. Coypu però non è una libreria implementata da zero, ma sfrutta le funzionalità di Selenium per definire un livello di astrazione superiore.

Selenium è un progetto non open-source ma gratuito, sviluppato da un team che in passato ha anche lavorato in sinergia con Google per sviluppare alcune estensioni della libreria WebDriver, utilizzate poi internamente dall'azienda di Mountain View per i test di applicazioni web. Rispetto a Coypu e Capybara, Selenium è disponibile in più linguaggi e implementa anche le funzionalità per la navigazione in applicazioni per Android e iOS.

Inoltre il team di Selenium ha sviluppato altri progetti, come Selenium IDE, un plugin per Firefox per la definizione di test registrando le azioni nel browser, Selenium Grid, per l'esecuzione parallela dei test in più browser e sistemi operativi e Selenium Remote Control, un'architettura client-server per l'esecuzione in remoto dei test.

	Open-Source	Gratuito	Wrapper	Mobile	Linguaggi
Capybara	✓	✓	✗	✗	Ruby
Coypu	✓	✓	✓	✓	C# Java
Selenium	✗	✓	✗	Tramite Selenium	C# Java Ruby Python PHP Perl JS

Tabella 5.8: Confronto riassuntiva fra le librerie.

#### Funzionalità

Le valutazioni sull'usabilità, leggibilità e manutenibilità della libreria dipendono dal linguaggio utilizzato. Ad esempio Capybara ha una sintassi molto più leggera e

comprensibile rispetto a Selenium anche grazie a Ruby e Coypu in C# sfrutta la migliore integrazione del linguaggio con funzioni e closures.

Strutturando in maniera diversa il proprio codice, Capybara è estremamente modulare, Selenium ha un'organizzazione classica suddivisa per package e funzionalità e Coypu tende ad raccogliere i metodi all'interno di un insieme ridotto di classi, tutte le librerie implementano con successo le funzionalità di base della navigazione web.

Le differenze significative riguardano le signature dei metodi. Coypu e Capybara prediligono definire delle funzionalità leggibile ed estendibili tramite parametri opzionali, come dizionari o la classe "Options" di Coypu. Selenium al contrario introduce una gerarchia di classi articolata e prevede diverse enumerazioni e classi factory per navigare nel browser.

Coypu e Capybara cercano di astrarre la navigazione dell'applicazione web mantenendo la stessa prospettiva dell'utente, introducendo diversi selettori sul contenuto testuale degli elementi e fornendo gli strumenti per definire espressioni regolari o stringhe parziali. Al contrario Selenium opta per un'implementazione più a basso livello, ad esempio non è presente il selettore sul contenuto testuale, e per l'utilizzo del pattern Page Object[71], nel seguito PO, teorizzato da Martin Fowler, fornendo alcuni metodi per favorirne l'implementazione.

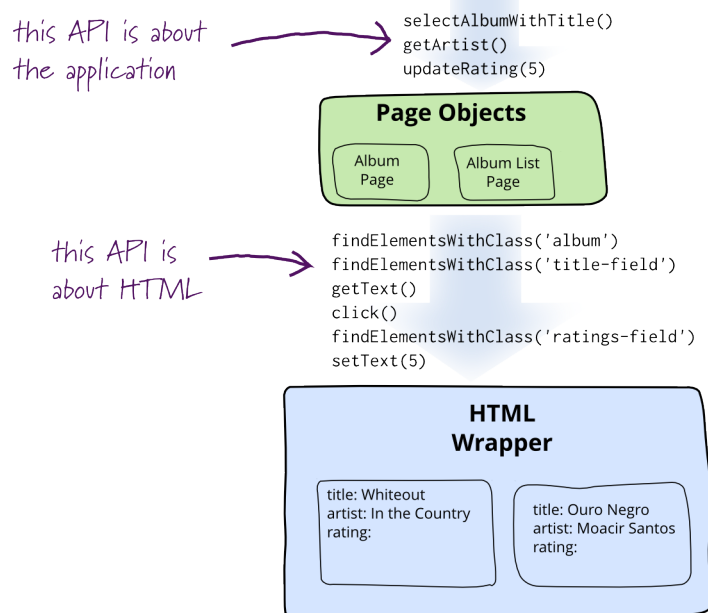


Figura 5.1: Implementazione del pattern Page Object.

Un PO è un'astrazione di una pagina web, o più in generale di qualsiasi interfaccia grafica, effettuata tramite il paradigma ad oggetti. All'interno della classe sono descritte, tramite dei metodi, le funzionalità offerte dalla pagina, utilizzate dai test per effettuare la navigazione. Il pattern permette di fattorizzare la definizione delle operazioni, rendendo il codice più mantenibile e rimuovendo dall'implementazione dei test ogni dettaglio a basso livello della pagina.

Selenium, e di conseguenza Coypu, sono meglio integrati con i browser web, fornendo le funzionalità per utilizzare componenti come la cronologia e i preferiti.

L'utilizzo del pattern PO è adatto allo sviluppo di progetti in ATDD, ma richiede di introdurre ulteriore complessità ai test. Al contrario, scegliendo di utilizzare le funzionalità ad alto livello di Capybara e Coypu l'implementazione è più rapida ed allo stesso tempo fragile.

	Form	Selettori	PO	Azioni
Capybara	✓	Si, alto livello	✗	✓
Selenium	✓	✓	✓	Si, integrazione con il browser
Coypu	✓	Si, alto livello	Si, tramite Selenium	Si, integrazione con il browser

Tabella 5.9: Funzionalità per la definizione dei test.

### Gestione del CSS

Per la verifica delle proprietà stilistiche degli elementi del DOM è possibile effettuare l'accesso al valore "style" oppure verificare che le regole definite nei fogli di stile siano applicate.

Tutte le librerie implementano dei meccanismi per accedere ai singoli attributi di un elemento HTML, Capybara e Coypu definiscono l'operatore "[ ]" per ottenere i singoli valori con la stessa interfaccia di un'array associativo, Selenium invece fornisce il metodo "getAttribute".

In Capybara l'accesso all'attributo "style" è l'unico metodo nativo per ottenere informazioni riguardo lo stile di un tag HTML. Però è possibile implementare script JQuery per ulteriori modalità d'accesso. Selenium invece implementa un metodo "getCssValue" per ottenere le proprietà stilistiche, sia che siano dichiarate nell'attributo "style" sia che siano attribuite tramite fogli di stile. Purtroppo però gli attributi utilizzabili sono solo quelli presenti nel CSS 2.

Coypu non implementa le funzionalità per ottenere le proprietà stilistiche di un elemento, se non utilizzando l'attributo "style", ma è comunque possibile utilizzare direttamente la libreria di Selenium ed utilizzare il metodo "getCssValue".



	Attributi inline	Fogli di stile
Capybara	Valore serializzato	✘
Selenium	Singolo valore, solo CSS2	Singolo valore, solo CSS2
Coypu	Singolo valore, tramite Selenium	Singolo valore, tramite Selenium

Tabella 5.10: Funzionalità per l'accesso alle proprietà stilistiche

Nonostante le limitazioni legate alla versione del CSS, Selenium è l'unica libreria ad implementare una funzionalità specifica per ottenere i valori delle proprietà estetiche di un elemento HTML.

### Asincronia

Per la gestione dei comportamenti asincroni sono state utilizzate due tecniche differenti:

- tramite attese implicite, implementate in tutte le librerie, si assume che ogni elemento sia potenzialmente il risultato di un'operazione asincrona ed è fissato un tempo massimo di attesa ed una frequenza di polling;
- tramite attese esplicite si attende che lo stato della pagina coincida con l'asserzione entro un certo tempo massimo.

Capybara e Coypu implementano il meccanismo delle attese implicite. Capybara però necessita di utilizzare il metodo "synchronize" per la gestione di comportamenti asincroni complessi (come il widget per l'auto-completamento, che richiede l'applicazione di regole di CSS per definire l'interfaccia grafica).

Selenium invece implementa entrambe i meccanismi, in particolare per SBlog sono state utilizzati le attese esplicite ed i metodi presenti nella libreria per la verifica dello stato della pagina.

Sia Selenium che Capybara però, quando ricercano un elemento all'interno del DOM, considerano anche gli elementi non visibili. Spesso i widget grafici utilizzano elementi nascosti nel DOM e l'impossibilità di configurare Selenium, evitando elementi visibili da un utente, complica l'utilizzo di widget asincroni. Capybara invece permette di specificare per ciascuna invocazione un parametro "visible" booleano.

Coypu invece adotta l'approccio opposto, ricercando nella pagina solo elementi visibili, mantenendo la stessa prospettiva di un utente durante la navigazione web ma permettendo comunque di configurare il comportamento. Verificando solo elementi visibili ed utilizzando le attese implicite, Coypu rappresenta la scelta migliore, soprattutto in applicazioni web moderne.

	Attese esplicite	Attese implicite	Semplicità nell'uso dei widget
Capybara	✗	✓	+
Selenium	✓	✓	- (con le attese esplicite) + (con le attese implicite)
Coypu	Tramite Selenium	✓	++

Tabella 5.11: Valutazione delle funzionalità per la verifica di comportamenti asincroni

### Manutenibilità

Utilizzando selettori legati alla struttura, sia che siano sul CSS o espressi tramite XPath, si facilita la definizione di test fragili. Per evitare costanti operazioni di modifica dei test è quindi opportuno utilizzare selettori più ad alto livello, come sul testo, soprattutto se, come in MVC5, è possibile fattorizzare le espressioni testuali.

Anche utilizzare una libreria di asserzioni ben definita può semplificare la manutenibilità, implementando test più leggibili ed espressivi.

La miglior libreria per definire asserzioni è il progetto “rspec-expectations” di RSpec, che permette di definire metodi compatti e leggibili attraverso Ruby, sfruttare la segnatura del metodo “expect” e la presenza di numerosi metodi per la definizione di condizioni, come “raise\_error” per la verifica di comportamenti eccezionali.

NUnit è leggermente inferiore a RSpec ma comunque una valida alternativa per implementare le asserzioni nei passi dei test di accettazione. Una delle caratteristiche più apprezzate è la possibilità di utilizzare il metodo “That” della classe “Assert” e i metodi factory della classe “Shows” per descrivere asserzioni facilmente leggibili.

JUnit, che utilizzato in isolamento è uno strumento completo, intuitivo e potente, con Cucumber non esprime a pieno le sue potenzialità. Non è, infatti, possibile utilizzare nessuna funzionalità esistente per la verifica delle eccezioni, costringendo l'utente a provvedere manualmente, sviluppando test eccessivamente complessi.

	Espressività	Leggibilità	Modularità della libreria
RSpec	++	++	++
NUnit	++	++	-
JUnit	+	+	-

Tabella 5.12: Valutazione delle librerie per l'implementazione delle asserzioni nei test.

**Conclusione**

Potendo scegliere lo stack di applicazioni per definire ed implementare i test di accettazione in maniera ideale consiglieri Cucumber, nella versione per Ruby, Coypu e RSpec.

Purtroppo Coypu non è utilizzabile in Ruby, per cui nelle scelte realistiche bisogna rassegnarsi ad avere elementi sub-ottimali nello stack. Gli strumenti utilizzati per sviluppare RBlog rappresentano la miglior soluzione: si integrano perfettamente fra loro e permettono di implementare test in maniera estremamente leggibile e in un numero ridotto di righe di codice.

Lo stack utilizzato per .NET ha in SpecFlow e nell'implementazione degli hook il suo elemento debole, trascurando quest'aspetto, i test scritti in Coypu e NUnit sono fra quelli più leggibili ed allo stesso tempo meno fragili.

Gli strumenti scelti per Java sono completi ma non eccellono a causa dei problemi di Selenium nella gestione degli eventi asincroni e di JUnit per la specifica delle asserzioni.

# xBlog

- Il codice sorgente di RBlog, il caso di studio sviluppato in Ruby e RoR, è consultabile su <https://github.com/TTia/rblog>;
- Il codice sorgente di SBlog, il caso di studio sviluppato in Spring e Java, è consultabile su <https://github.com/TTia/sblog>;
- Il codice sorgente di CSBlog, il caso di studio sviluppato in ASP.NET MVC5 e C#, è consultabile su <https://github.com/TTia/csblog>;

# Acronimi

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

ATDD: Acceptance Test-Driven Development	JPA: Java Persistence API
BDD: Behavior-Driven Development	JPQL: Java Persistence Query Language
CRUD: Create, Read, Update, Delete	JSP: JavaServer Page
CSS: Cascading Style Sheets	LINQ: Language-Integrated Query
CVS: Concurrent Versions System	MVC: Model-View-Controller
DOM: Document Object Model	POJO: Plain Old Java Object
EF: Entity Framework	REST: REpresentational State Transfer
ERB: Embedded RuBy	RoR: Ruby on Rails
GUI: Graphical User Interface	SSH: Secure SHell
HTML: HyperText Markup Language	STS: Spring Tool Suite
HTTP: HyperText Transfer Protocol	TDD: Test-Driven Development
IDE: Integrated Development Environment	URL: Uniform Resource Locator
IRB: Interactive Ruby Shell	VS: Visual Studio

# Bibliografia

- [1] <http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4655401&url=http%3A%2F%2Fieeexplore.ieee.org/> 1
- [2] <https://www.ruby-lang.org/it/> 9
- [3] <http://ruby-doc.org/core-2.1.4/Symbol.html> 11
- [4] <http://www.ruby-doc.org/core-2.1.4/String.html> 11
- [5] <https://www.ruby-lang.org/en/news/2014/09/18/ruby-2-2-0-preview1-released/> 11
- [6] <http://www.ruby-doc.org/core-2.1.4/Hash.html> 11
- [7] <http://www.scribd.com/doc/27174770/Garbage-Collection-and-the-Ruby-Heap> 11
- [8] <http://www.ruby-doc.org/core-2.1.4/Object.html> 11
- [9] <http://www.ruby-doc.org/core-2.1.4/BasicObject.html> 11
- [10] <http://rubykoans.com/> 13
- [11] <https://rubymonk.com/> 13
- [12] <https://www.jetbrains.com/ruby/> 14
- [13] <http://www.martinfowler.com/eaCatalog/activeRecord.html> 15
- [14] [http://guides.rubyonrails.org/action\\_controller\\_overview.html](http://guides.rubyonrails.org/action_controller_overview.html) 15
- [15] <http://api.rubyonrails.org/classes/ActionController/Base.html> 15
- [16] <http://guides.rubyonrails.org/routing.html> 16
- [17] [http://guides.rubyonrails.org/action\\_controller\\_overview.html#filters](http://guides.rubyonrails.org/action_controller_overview.html#filters) 16
- [18] <http://en.wikipedia.org/wiki/ERuby>
- [19] [http://guides.rubyonrails.org/layouts\\_and\\_rendering.html#understanding-yield](http://guides.rubyonrails.org/layouts_and_rendering.html#understanding-yield) 17

- [20] <http://en.wikipedia.org/wiki/ERuby>
- [21] <http://api.rubyonrails.org/classes/ActionView/Helpers/UrlHelper.html> 18
- [22] <http://guides.rubyonrails.org/testing.html> 18
- [23] <https://github.com/rspec/rspec> 18
- [24] <https://github.com/seattlerb/minitest> 18
- [25] [https://github.com/DatabaseCleaner/database\\_cleaner](https://github.com/DatabaseCleaner/database_cleaner) 18
- [26] [https://github.com/thoughtbot/factory\\_girl](https://github.com/thoughtbot/factory_girl) 18
- [27] <http://guides.rubyonrails.org/testing.html##rake-tasks-for-running-your-tests> 19
- [28] <https://rubygems.org/> 19
- [29] <http://maven.apache.org/> 19
- [30] <http://www.gradle.org/> 19
- [31] <http://bundler.io/gemfile.html> 19
- [32] <https://github.com/rails/spring> 19
- [33] <http://git-scm.com/docs/git-revert> 19
- [34] <http://guides.rubyonrails.org/migrations.html> 20
- [35] <http://api.rubyonrails.org/classes/ActionView/PartialRenderer.html> 20
- [36] <http://www.yaml.org/> 22
- [37] <http://www.postgresql.org/> 22
- [38] <http://cukes.info/> 23
- [39] <http://www.agilemodeling.com/artifacts/userStory.htm> 23
- [40] 3
- [41] <https://github.com/cucumber/cucumber/wiki/Gherkin> 3
- [42] <http://jnicklas.github.io/capybara/> 25
- [43] <https://github.com/jnicklas/capybara#drivers> 26
- [44] <http://phantomjs.org/> 26
- [45] <https://github.com/teampoltergeist/poltergeist> 26
- [46] <https://developer.chrome.com/devtools> 26

- [47] <https://www.webkit.org/> 26
- [48] [http://en.wikipedia.org/wiki/Ajax\\_\(programming\)](http://en.wikipedia.org/wiki/Ajax_(programming)) 27
- [49] <http://www.rubydoc.info/github/jnicklas/capybara/Capybara/Node> 28
- [50] <http://www.rubydoc.info/github/jnicklas/capybara/Capybara/Node> 28
- [51] <https://www.relishapp.com/cucumber/cucumber/docs/transforms> 29
- [52] <https://github.com/rspec/rspec-core> 29
- [53] <https://github.com/rspec/rspec-expectations> 30
- [54] <https://github.com/rspec/rspec-mocks> 30
- [55] <https://github.com/rspec/rspec-rails> 30
- [56] <http://guides.rubyonrails.org/testing.html> 30
- [57] <https://github.com/metaskills/minitest-spec-rails> 30
- [58] <http://sass-lang.com/> 32
- [59] <http://en.wikipedia.org/wiki/XPath> 36
- [60] <https://github.com/cucumber/cucumber/wiki/Hooks> 38
- [61] <https://github.com/codahale/bcrypt-ruby> 40
- [62] <http://jqueryui.com/> 44
- [63] <http://www.martinfowler.com/bliki/POJO.html> 50
- [64] <http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/testing.html> 61
- [65] <http://junit.org/> 61
- [66] <http://junit.sourceforge.net/javadoc/org/junit/runner/RunWith.html> 61
- [67] <https://github.com/cucumber/cucumber-eclipse> 64
- [68] <http://selenium.googlecode.com/git/docs/api/java/org/openqa/selenium/package-tree.html> 65
- [69] <https://github.com/junit-team/junit/wiki/Rules> 74
- [70] <http://www.railsonmaui.com/tips/rails/capybara-phantomjs-poltergeist-rspec-rails-tips.html> 109
- [71] <http://martinfowler.com/bliki/PageObject.html> 122
- [72] <http://gatling.io/> 108



- [73] <https://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplersr2> 49
- [74] <http://spring.io/tools> 49
- [75] <http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html> 50
- [76] <https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html> 51
- [77] <http://eclipse.org/eclipselink/> 50
- [78] <http://www.mysql.it/> 52
- [79] <http://en.wikipedia.org/wiki/InnoDB> 52
- [80] <http://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository> 53
- [81] [http://en.wikipedia.org/wiki/Java\\_Persistence\\_Query\\_Language](http://en.wikipedia.org/wiki/Java_Persistence_Query_Language) 54
- [82] <https://docs.oracle.com/javaee/6/tutorial/doc/gjtv.html> 54
- [83] <http://www.thymeleaf.org/> 58
- [84] <http://www.oracle.com/technetwork/java/javaee/jsp/index.html> 58
- [85] <https://docs.oracle.com/javaee/7/api/javax/validation/constraints/package-summary.html> 60
- [86] <https://github.com/cucumber/cucumber-jvm> 63
- [87] <http://picocontainer.codehaus.org/> 63
- [88] <http://www.seleniumhq.org/> 64
- [89] <https://github.com/detro/ghostdriver> 65
- [90] [http://docs.seleniumhq.org/docs/03\\_webdriver.jsp#cookies](http://docs.seleniumhq.org/docs/03_webdriver.jsp#cookies) 76
- [91] [http://docs.seleniumhq.org/docs/04\\_webdriver\\_advanced.jsp#webdriver-advanced-usage](http://docs.seleniumhq.org/docs/04_webdriver_advanced.jsp#webdriver-advanced-usage) 77
- [92] <http://www.asp.net/mvc/mvc5> 79
- [93] <http://www.specflow.org/> 79
- [94] <https://github.com/featurist/coypu> 79, 86
- [95] <http://www.nunit.org/> 86
- [96] <https://www.nuget.org/> 79
- [97] <https://entityframework.codeplex.com/wikipage?title=specs> 80

- [98] [http://msdn.microsoft.com/it-it/library/system.data.entity.dbcontext\(v=vs.113\).aspx](http://msdn.microsoft.com/it-it/library/system.data.entity.dbcontext(v=vs.113).aspx) 80
- [99] <http://msdn.microsoft.com/it-it/library/bb397926.aspx> 81
- [100] [http://msdn.microsoft.com/it-it/library/system.linq.iqueryable\(v=vs.110\).aspx](http://msdn.microsoft.com/it-it/library/system.linq.iqueryable(v=vs.110).aspx) 81
- [101] [http://msdn.microsoft.com/it-it/library/system.linq.iqueryable\(v=vs.110\).aspx](http://msdn.microsoft.com/it-it/library/system.linq.iqueryable(v=vs.110).aspx) 81
- [102] [http://www.asp.net/web-pages/overview/getting-started/introducing-razor-syntax-\(c\)](http://www.asp.net/web-pages/overview/getting-started/introducing-razor-syntax-(c)) 83
- [103] [http://msdn.microsoft.com/it-it/library/dd410269\(v=vs.100\).aspx](http://msdn.microsoft.com/it-it/library/dd410269(v=vs.100).aspx) 83
- [104] <http://getbootstrap.com/>

*« Fletto i muscoli e sono nel vuoto! »*